

Meus Estudos em Análise e Desenvolvimento de Sistemas

PUC-MG

BRUNO DE M. RUAS

24 de maio de 2022

Conteúdo

I	Implementação de Sistemas de Software	1
1	Projeto: Desenvolvimento Web Front-End	2
1.1	Etapa 1	2
1.2	Etapa 2	2
1.3	Etapa 3	3
1.4	Etapa 4	3
1.5	Etapa 5	3
2	Algoritmos e Abstração de Dados	5
2.1	Estrutura de Dados Homogêneas e Heterogêneas	6
2.1.1	Estrutura de Dados Homogêneas	6
2.1.2	Estrutura de Dados Heterogêneas	6
2.2	Tipos Abstratos de Dados - Classes - Implementação	6
2.2.1	Definição de um TAD - Classes e Objetos	6
2.2.2	Atributos, Propriedades e Métodos de Classe	6
2.2.3	Mecanismos de Visibilidade/Acessibilidade	6
2.2.4	Construtores de Classe	6
3	Algoritmos e Lógica de Programação	7
3.1	Lógica de Programação e Estrutura de Controle, Funções e Procedimentos	8
3.1.1	Conceito de Algoritmo	8
3.1.2	Variáveis	8
3.1.3	Etapas de um Algoritmo e Operador de Atribuição	10
3.1.4	Estrutura Sequencial	12
3.1.5	Estrutura Condicional	16
3.1.6	Estrutura de Repetição	21
3.2	Manipulação de Dados em Memória Primária e Secundária	26
3.2.1	Criando e Usando Funções e Procedimentos	27
3.2.2	Passagem de Parâmetros	28
3.2.3	Manipulação de Arquivos em C#	28

4	Desenvolvimento Web Front-End	29
4.1	A Web: Evolução, Padrões e Arquitetura	30
4.1.1	Histórico e Evolução da Web	30
4.1.2	W3C e os Padrões da Web	30
4.1.3	Componentes da Arquitetura da Web	31
4.1.4	URI, URL e URN	32
4.1.5	Protocolo HTTP	32
4.1.6	Servidores Web	34
4.1.7	Dinâmica de Aplicações Web	36
4.2	Desenvolvimento de Interfaces Web	37
4.2.1	A Linguagem HTML	37
4.2.2	A Linguagem CSS	48
4.2.3	A Linguagem JavaScript	69
5	Fundamentos de Engenharia de Software	101
5.1	Conceitos e Processos de Software	102
5.1.1	Definições	102
5.1.2	Modelos e Princípios de Processo de Software	102
5.1.3	Processos Ágeis	103
5.1.4	Processos Prescritivos	105
5.1.5	Quando usar cada Processo?	106
5.1.6	Requisitos	107
5.1.7	Requisitos Funcionais	107
5.1.8	Requisitos Não Funcionais	107
5.2	Atividades e Artefatos da Engenharia de Software	108
5.2.1	Atividades Técnicas	109
5.2.2	Atividades Gerenciais	110
5.2.3	Testes de Software	111
5.2.4	Artefatos e Templates	112
5.2.5	Desenhando Processos de Software	114
6	Lógica Computacional	116
6.1	Bibliografia	116
6.2	Pensamento Lógico	117
6.2.1	Introdução	117
6.2.2	O que é Lógica?	117
6.2.3	Motivação	117
6.2.4	Definições	117
6.2.5	Subconjuntos	117
6.2.6	Operações sobre Conjuntos	117
6.2.7	Princípios da Lógica Proposicional	117
6.2.8	Conectivos Lógicos	117
6.2.9	Tabela Verdade e Equivalência Lógica	117
6.2.10	Predicados e Quantificadores	117

6.2.11	Ligando Variáveis	117
6.2.12	Negações	117
6.3	Pensamento Analítico	117
6.3.1	Provas de Teoremas	117
6.3.2	Regras de Inferência	117
6.3.3	Argumentos Válidos	117
6.3.4	Indução Matemática	117
6.3.5	Indução Forte	117
6.3.6	Recursão	117
6.3.7	Especificação de Sistemas	117
6.3.8	Verificação de Programas	117
7	Matemática Básica	118
8	Organização de Computadores	120
8.1	Fundamentos de Organização de Computadores	121
8.1.1	Representação de Dados e Sistemas Binário	121
8.1.2	Conceitos de Lógica Digital	122
8.1.3	Circuitos Lógicos Digitais Básicos	125
8.1.4	Introdução à Organização de Computadores	125
8.1.5	Unidade Central de Processamento - UCP	125
8.1.6	Memória	125
8.1.7	Entrada e Saída	125
8.2	Arquitetura de Computadores	125
8.2.1	Arquiteturas RISC e CISC	125
8.2.2	Arquitetura do Conjunto de Instruções: Exemplo do MIPS	125
8.2.3	Linguagem de Montagem	125
8.2.4	Conceito de Pipeline de Instruções	125
8.2.5	Paralelismo em Nível de Instruções	125
8.2.6	Paralelismo em Nível de Processadores	125
9	Pensamento Computacional	126
9.1	Conceitos e Competências de Pensamento Computacional . .	127
9.2	Computação Desplugada	128
II	Análise e Projeto de Software	129
10	Projeto: Desenvolvimento de uma Aplicação Interativa	130
11	Algoritmos e Estrutura de Dados	131
12	Desenvolvimento Web Back-End	132

13 Design de Interação	133
14 Engenharia de Requisitos de Software	134
15 Fundamentos de Redes de Computadores	135
16 Manipulação de Dados com SQL	136
17 Modelagem de Dados	137
18 Programação Modular	138
 III Processo de Negócio e Desenvolvimento de Software	 139
19 Projeto: Desenvolvimento de uma Aplicação Móvel em um Ambiente de Negócio	140
20 Desenvolvimento de Aplicações Móveis	141
21 Estatística Descritiva	142
22 Gerência de Configuração	143
23 Gerência de Projetos de TI	144
24 Gerência de Requisitos de Software	145
25 Qualidade de Processos de Software	146
 IV Infraestrutura para Sistemas de Software	 147
26 Projeto: Desenvolvimento de um Aplicação Distribuída	148
27 APIs e Web Services	149
28 Arquitetura de Software Distribuído	150
29 Banco de Dados NoSQL	151
30 Cloud Computing	152
31 Projeto de Software	153
32 Teste de Software	154

V Empreendedorismo e Inovação com Sistemas de Software	155
33 Projeto: Desenvolvimento de um Sistema Sociotecnológico Inovador	156
34 Compliance em TI	157
35 Computadores e Sociedade	158
36 Empreendedorismo e Inovação	159
37 Implantação de Soluções de TI	160
38 Segurança Aplicada ao Desenvolvimento de Software	161

Parte I

Implementação de Sistemas de Software

Capítulo 1

Projeto: Desenvolvimento Web Front-End

1.1 Etapa 1

Objetivo da Etapa: Definir o problema a ser solucionado e os componentes do seu grupo de trabalho. Nesta etapa você entregará como tarefa dois artefatos: a **documentação de contexto** e a **especificação do projeto**.

O template da documentação do projeto pode ser baixado nesse [LINK](#)

Microfundamentos a serem estudados:

- Matemática Básica
- Pensamento Computacional
- Fundamentos de Engenharia de Software

1.2 Etapa 2

Objetivo da Etapa: Projetar a interface da aplicação e a arquitetura da solução, além de definir o ambiente de trabalho que será utilizado pela equipe para desenvolver o projeto. Os artefatos a serem produzidos são: **Projeto de Interface**, **Metodologia** e **Arquitetura da Solução**.

Microfundamentos a serem estudados:

- Fundamentos de Engenharia de Software
- Desenvolvimento Web Front-End
- Lógica Computacional

1.3 Etapa 3

Objetivo da Etapa: Desenvolver a homepage e, pelo menos, uma funcionalidade da solução projetada. O primeiro artefato a ser gerado é o **Template do site**, que determina o layout padrão do site (HTML e CSS) que será utilizado em todas as páginas com a definição de identidade visual, aspectos de responsividade e iconografia. No desenvolvimento das funcionalidades, cada artefato gerado (código fonte) deve estar relacionado a um requisito funcional e/ou não funcional.

Microfundamentos a serem estudados:

- Desenvolvimento Web Front-End
- Algoritmos e Lógica de Programação

1.4 Etapa 4

Objetivo da Etapa: Finalizar o desenvolvimento da solução e irá elaborar e executar o plano de testes funcionais. Os artefatos serão: O **plano de testes de software** e O **Registro de Testes de Software**.

Microfundamentos a serem estudados:

- Algoritmos e Lógica de Programação
- Fundamentos de Engenharia de Software
- Algoritmos e Abstração de Dados

1.5 Etapa 5

Objetivo da Etapa: Apresentar a versão final da solução implantada.

A apresentação do projeto consiste na geração de um conjunto de slides em um arquivo no formato ppt, pptx ou pdf, contemplando os seguintes itens:

- Contexto (Problema, Público-alvo);
- Requisitos;
- Solução Implementada (funcionalidades de software);
- Conclusão da elaboração do projeto (pontos positivos, desafios, aprendizado).

CAPÍTULO 1. PROJETO: DESENVOLVIMENTO WEB FRONT-END4

Recomenda-se não ultrapassar 10 slides, pois o tempo de apresentação é limitado a 10 minutos, sendo 5 minutos para o projeto (slides) e 5 minutos para a demonstração da aplicação.

A equipe também deverá gravar um vídeo de, no máximo, três minutos, com a apresentação da solução. Vocês deverão abrir a aplicação hospedada e apresentar o seu funcionamento.

Microfundamentos a serem estudados:

- Fundamentos de Engenharia de Software
- Organização de Computadores

Capítulo 2

Algoritmos e Abstração de Dados

Bibliografia

Bibliografia Básica

- ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. Fundamentos da programação de computadores. São Paulo: Pearson, 2012. ISBN 9788564574168
- SOUZA, Marco A. Furlan de; GOMES, Marcelo Marques; SOARES, Marcio Vieira; CONCÍLIO, Ricardo. Algoritmos e lógica de programação: um texto introdutório para a engenharia. São Paulo: Cengage Learning, 2019. ISBN: 9788522128150
- ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS. New York: Association for Computing Machinery.,1979-. 6 times a year. Absorvido ACM letters on programming languages and systems. ISSN 0164-0925. Disponível em: <https://dl-acm-org.ez93.periodicos.capes.gov.br/citation.cfm?id=J783>
- AGUILAR, Luis Joyanes. Fundamentos de programação algoritmos, estruturas de dados e objetos. 3. ed. Porto Alegre: AMGH, 2008. ISBN: 9788580550146

Bibliografia Complementar

- DEITEL, Harvey M; DEITEL, Paul J. Java - como programar. 8. ed. São Paulo: Pearson, 2010. ISBN 9788576055631
- GRIFFITHS, Ian. Programming C# 8.0. O'Reilly Media, Inc. 2019. ISBN 9781492056812

- MANZANO, José Augusto N. G; OLIVEIRA, Jayr Figueiredo de. Algoritmos: lógica para desenvolvimento de programação de computadores. 28. ed. rev. e atual. São Paulo, SP: Érica, 2016. E-book. ISBN 9788536518657
- PROGRAMMING AND COMPUTER SOFTWARE. New York, Consultants Bureau, 1975-. Bimestral. ISSN 1608-3261. Disponível em: <https://link-springer-com.ez93.periodicos.capes.gov.br/journal/volumesAndIssues/11086>
- PRICE, Mark J. C# 8.0 and .NET Core 3.0 - Modern Cross - Platform Development. O'Reilly Media; 2019. ISBN 9781788478120
- PUGA, Sandra; RISSETTI, Gerson. Lógica de programação e estruturas de dados com aplicações em Java. 2. ed. São Paulo: Prentice Hall, 2009. ISBN 9788576052074

2.1 Estrutura de Dados Homogêneas e Heterogêneas

2.1.1 Estrutura de Dados Homogêneas

2.1.2 Estrutura de Dados Heterogêneas

2.2 Tipos Abstratos de Dados - Classes - Implementação

2.2.1 Definição de um TAD - Classes e Objetos

2.2.2 Atributos, Propriedades e Métodos de Classe

2.2.3 Mecanismos de Visibilidade/Acessibilidade

2.2.4 Construtores de Classe

Capítulo 3

Algoritmos e Lógica de Programação

Bibliografia

Bibliografia Básica

- Ana Fernanda Gomes ASCENCIO; Edilene Aparecida Veneruchi de CAMPOS. Fundamentos da Programação de Computadores: algoritmos, Pascal, C/C++ e Java - 2ª edição. São Paulo, SP : Pearson Education do Brasil, 2012

Bibliografia Complementar

- H. DEITEL et. Al. C#: Como Programar. São Paulo: Makron Books, 2003
- John SHARP. Microsoft Visual C# 2013. Grupo A, 2014
- André Luiz Villar FORBELLONE, Henri Frederico EBERSPÄCHER. Lógica de programação: a construção de algoritmos e estruturas de dados. São Paulo: Prentice Hall, 2005.
- MANZANO, José Augusto N. G; OLIVEIRA, Jayr Figueiredo de. Algoritmos: lógica para desenvolvimento de programação de computadores 28. ed. rev. e atual. São Paulo, SP: Érica, 2016
- Sandra PUGA, Gerson RISSETTI. Lógica de Programação e Estrutura de Dados: com aplicações em Java - 2ª edição. São Paulo : Pearson, 2017

3.1 Lógica de Programação e Estrutura de Controle, Funções e Procedimentos

3.1.1 Conceito de Algoritmo

Em resumo, um algoritmo é uma sequência de ordens que, se seguidas, deve gerar um resultado previsto e desejado. Não vale muito a pena aprofundar além disso agora.

3.1.2 Variáveis

Uma variável é uma posição na memória do computador. Isso mesmo, é algo com endereço definido, nada solto no universo ou obscuro dentro da máquina. Esse dado é gravado para posterior leitura por parte do programa que está sendo executado.

Podemos ter como origem do dado o próprio programa que está sendo executado ou o usuário por meio da entrada de dados ou, ainda, a leitura de dados previamente armazenados na memória do computador.

Cada variável possui vários tipos de atributos que podemos elencar na lista abaixo:

- Tem um endereço na memória
- Possui um nome de identificação
- Armazena um valor
- Possui um tipo de dados
 - Numérico - Inteiros e Reais (Double ou Ponto Flutuante)¹
 - Não numérico - Lógico e Caractere
 - Arranjos - Vetores e Matrizes
 - Arquivos
 - etc

O endereço da memória é feita pelos endereços que possuem apenas 2 estágios de registro: 0 ou 1. Um bit é exatamente essa medida de registro. 1 byte é composto de 8bits. 1 kilobyte contém 1024bytes (2^{10}). 1 megabyte contém 1024KB (2^{20}). 1 gigabyte contém 1024MB (2^{30}). 1 terabyte possui 1024GB (2^{40}).

Ou seja, se um computador possui 3 gigas de memória RAM, ele tem 3 bilhões de bytes como endereços disponíveis para um registro na memória².

¹Float possui precisão simples e Double possui dupla precisão.

²Isso é muito impressionante!

Cada byte possui um endereço único. Na memória RAM costuma-se usar a base Hexadecimal para definir os endereços. Vamos ver isso melhor na matéria de Arquitetura de Computadores.

Também vimos que as variáveis possuem identificadores. Em `c#` existem regras para a criação dos identificadores:

- Devem começar com uma letra
- Não podem ter espaços
- Não podem usar uma das palavras reservadas pela linguagem
- É case sensitive, ou seja, o nome "Var" é diferente do nome "var"

O identificador deve sempre ter algum sentido que permita a rápida interpretação por parte de algum leitor do código fonte da aplicação. Isso é fundamental para manutenção de códigos produzidos por várias pessoas diferentes. Entretanto, devemos evitar nomes de variáveis muito grandes. Na verdade, quanto menor o nome, mantido o sentido na leitura, melhor.

Outra boa prática é usar o camelCase para variáveis com mais de uma palavra. Esse padrão é bem simples: primeira palavra em minúsculo e a segunda com a primeira letra em maiúscula.

Declaração de Variáveis em `c#`

Vamos aprender agora como declarar os 4 tipos mais simples de variáveis:

- `int` para definir inteiros
- `double` ou `float` para definir números reais
- `string` para cadeia de caracteres
- `bool` para as variáveis booleanas (`true` ou `false`)

```
static void Main(string[] args)
{
    int idade, numero;
    double peso, salario;
    string nomePai, rua, dtNasc;
    bool temCasa;
}
```

Nesse código acima temos a declaração de várias variáveis de cada tipo. Basta colocar o tipo da variável seguido do nome de cada uma delas separadas por vírgula.

Comentário: Em `c#` todas as linhas precisam conter um caracter de encerramento que é o ponto e vírgula `;`. Sem esse token, o computador vai interpretar a próxima linha como sendo a continuação da anterior.

3.1.3 Etapas de um Algoritmo e Operador de Atribuição

Podemos resumir um algoritmo simples como contendo apenas 3 etapas: 1) A entrada de dados; 2) O processamento e 3) A saída de dados.

Na etapa de processamento, podemos precisar de grande criatividade e esforço para produzir a saída de dados desejada. Como ferramenta para alcance desse objetivo, temos as **estruturas básicas** que são:

- Estrutura Sequencial - Usada para garantir a ordem correta dos passos
- Estrutura Condicional - Usada para permitir contexto ou cenários
- Estrutura de Repetição - Usada para evitar repetir código sequencial

Operador de Atribuição

Uma vez que já sabemos como atribuir uma variável ao seu tipo, precisamos aprender a como atribuir um valor a essas variáveis. Em `c#` a atribuição de valor é feita do seguinte modo:

```
idade = 28;  
numero = 9992233;
```

É possível fazer 3 tipos de atribuição de valor para uma variável: Valor fixo; Conteúdo de outra variável e Expressão aritmética ou booleana. Abaixo temos 3 exemplo disso.

```
idade = 28; // Valor fixo  
numero = idade; // Valor de outra variavel  
idadeNum = idade + numero; // Valor por expressão  
idadeEqNum = idade == numero; // Valor por expressao booleana
```

Nesse ponto vale uma reflexão. Será que sempre precisamos separar as etapas de atribuição do tipo de dados e do valor? Ou podemos, no mesmo momento, definir o tipo de dados e o valor da variável? A resposta, para nossa sorte, é que é possível em `c#` atribuição de tipo e valor ao mesmo tempo. A sintaxe fica desse modo:


```
int idade = 28;
int numero = idade;
double = 2.3;
string nome = "bruno";
int idadeNum = idade + numero;
bool idadeEqNum = idade == numero;
```

Saída de Dados

Agora que fizemos esse pequeno desvio no assunto, vamos retornar para as estruturas. Por incrível que pareça, é melhor a gente começar pelo final. A etapa de saída de dados.

Em `c#` temos duas maneiras de saída de dados:

```
// Imprime a informacao e cursor fica na mesma linha
Console.Write('string');

// Imprime a info mas o curso vai pra proxima linha
Console.WriteLine('string ' + var1 + ' string');
```

Nesses exemplos a gente pode perceber que podemos concatenar textos e variáveis textuais por meio do operador de soma. Além dessa forma, podemos fazer uso de PlaceHolders ou Interpolação de string como nos exemplos abaixo.

```
string nome = "Bruno";

// Concatenacao
Console.WriteLine("Meu nome é " + nome + ", obrigado!");

// Placeholder
Console.WriteLine("Meu nome é {0}, obrigado!", nome);

// Interpolacao
Console.WriteLine($"Meu nome é {nome}, obrigado!");
```

Os três exemplos geram o mesmo resultado mas são consideravelmente diferentes no método. Podemos usar o que for mais agradável para uma leitura do código e posterior manutenção do mesmo.

Entrada de Dados

Agora que sabemos como fazer nosso programa imprimir resultados, vamos aprender como inputar dados nele. Em c# podemos inserir dados com o seguinte comando:

```
variavel = Console.ReadLine();
```

O input captado por esse comando **sempre retorna uma string**. Mas as vezes precisamos nos certificar que o input foi feito da maneira correta. Para isso podemos manipular os dados inseridos do seguinte modo:

```
// Valores Inteiros
variavel = int.Parse(Console.ReadLine());
variavel = Convert.ToInt32(Console.ReadLine());

// Valores Reais
variavel = double.Parse(Console.ReadLine());
variavel = float.Parse(Console.ReadLine());
variavel = Convert.ToDouble(Console.ReadLine());

// Valores Booleanos
variavel = bool.Parse(Console.ReadLine());

// Valores String
variavel = Console.ReadLine();
```

Depois que executamos um dessas maneiras de coletar os dados, é de boa prática fornecermos um output logo após a entrada do dado. Isso evita aquela sensação de dúvida se o programa está sendo executado ou não. Ou seja, é bom sempre manter a dupla `ReadLine()` com o `WriteLine()`. O nome que damos para essa dupla é "Prompt".

Outra dica boa é usar o comando `Console.ReadKey()` no final do programa. Isso faz com que o console aguarde alguma tecla para finalizar. Evita que a aplicação abra e feche sem que o user consiga enxergar o resultado do processo.

3.1.4 Estrutura Sequencial

Agora que sabemos como receber valores do usuário e como devolver nova informação a partir do processamento, vamos começar a estudar a etapa de processamento através do estudo das estruturas sequenciais.

Operadores e Funções Aritméticas

Em `c#` nós temos os seguintes tokens para realizar as operações aritméticas:

- `+` Soma
- `-` Subtração
- `*` Multiplicação
- `/` Divisão
- `%` Resto da Divisão inteira

Além disso, temos a presença do que podemos chamar de **operadores de atribuição combinada**. Que faz a atribuição de valor com alguma operação aritmética:

- `soma += 100` é igual a `soma = soma + 100`
- `subtracao -= 10` é igual a `subtracao = subtracao - 10`
- `multiplica *= 2` é igual a `multiplca = multiplca * 2`
- `divide /= 3` é igual a `divide = divide / 3`
- `resto %= 2` é igual a `resto = resto / 2`

Não bastando essa grande variedade, temos os operadores de **incremento e decremento**:

- Pré-incremento (`++x`) - Usa `x + 1` antes do processamento
- Pós-incremento (`x++`) - Usa `x + 1` após o processamento
- Pré-decrécimo (`--x`) - Usa `x - 1` antes do processamento
- Pós-decrécimo (`x--`) - Usa `x -1` após o processamento

Essa diferença de pre e pós é importante para as estruturas de repetição, porque podemos lidar com índices de tabelas que começam com valores diferentes. Mas, no geral, devemos olhar cada caso para escolher o que usar.

Existem outras operações que estão contidas em um objeto nativo chamado **Math**. Para fazermos potenciação usamos o método `Math.Pow()`. Para fazermos raiz quadrada podemos usar `Math.Sqrt()`³.

³Embora a gente saiba que a radiciação é uma potência de fração.

Expressões Aritméticas

Agora podemos juntar tudo que vimos em um programa simples de cálculo da média de 5 valores. O código e o seu resultado podem ser vistos abaixo.

```
// Declaracao dos tipos das variaveis
int n1, n2, n3, n4, n5, soma;
float media;

// Input dos dados
Console.WriteLine("Programa para cálculo da média de 5 valores.");

Console.WriteLine("Por favor, forneça o primeiro número");
n1 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Por favor, forneça o segundo número");
n2 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Por favor, forneça o terceiro número");
n3 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Por favor, forneça o quarto número");
n4 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Por favor, forneça o quinto número");
n5 = Convert.ToInt32(Console.ReadLine());

// Processamento
soma = n1 + n2 + n3 + n4 + n5;
media = soma / 5;

// Output dos dados
Console.WriteLine("Os números fornecidos foram:");
Console.WriteLine("N1={0},N2={1},N3={2},N4={3},N5={4}", n1, n2, n3, n4, n5);
Console.WriteLine("Cuja média é igual a {0}", media);
```

```

1 Programa para cálculo da média de 5 valores.
2 Por favor, forneça o primeiro número
3 10
4 Por favor, forneça o segundo número
5 3
6 Por favor, forneça o terceiro número
7 4
8 Por favor, forneça o quarto número
9 70
0 Por favor, forneça o quinto número
1 2
2 Os números fornecidos foram:
3 N1 = 10,N2 = 3,N3 = 4,N4 = 70,N5 = 2
4 Cuja média é igual a 17
5

```

Figura 3.1: Programa de cálculo da média de 5 valores em c#

Outro exemplo que podemos usar é o do cálculo das raízes de uma função de segundo grau⁴. O programa abaixo recebe 3 valores e calcula as raízes,

⁴Se você não se lembra como resolver um problema desse, seu professor da sexta série está rindo de você nesse exato minuto.

a imagem logo após nos mostra o resultado no prompt de comando.

```
// Programa para calcular o valor de x em uma equacao de segundo grau

// Declaracao das variaveis
double a, b, c, x1, x2;

Console.WriteLine("Vamos resolver uma equação do tipo  $ax^2 + bx + c = 0$ ");

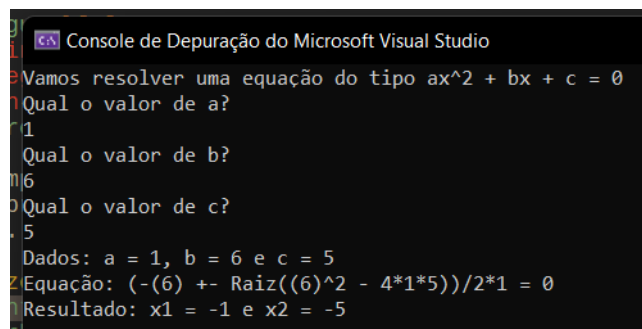
// Input das variaveis
Console.WriteLine("Qual o valor de a? ");
a = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("Qual o valor de b? ");
b = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("Qual o valor de c? ");
c = Convert.ToDouble(Console.ReadLine());

x1 = (-b + Math.Sqrt(Math.Pow(b,2) - 4 * a * c)) / 2 * a;
x2 = (-b - Math.Sqrt(Math.Pow(b, 2) - 4 * a * c)) / 2 * a;

// Output dos resultados
Console.WriteLine("Dados: a = {0}, b = {1} e c = {2}", a, b, c);
Console.WriteLine($"Equação:  $(-({b}) \pm \text{Raiz}(({b})^2 - 4*{a}*{c}))/2*{a} = 0$ ");
Console.WriteLine("Resultado: x1 = {0} e x2 = {1}", x1, x2);
```



```

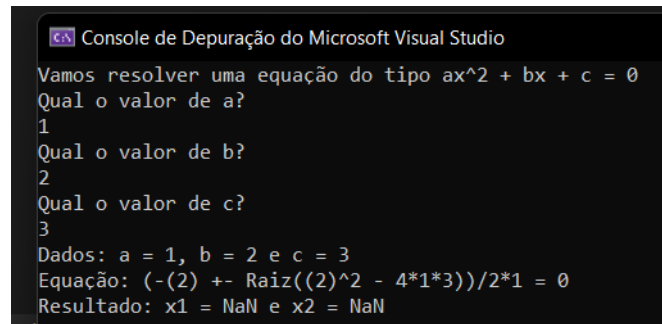
c# Console de Depuração do Microsoft Visual Studio
Vamos resolver uma equação do tipo  $ax^2 + bx + c = 0$ 
Qual o valor de a?
1
Qual o valor de b?
6
Qual o valor de c?
5
Dados: a = 1, b = 6 e c = 5
Equação:  $(-6) \pm \text{Raiz}((6)^2 - 4*1*5))/2*1 = 0$ 
Resultado: x1 = -1 e x2 = -5

```

Figura 3.2: Programa de resolução de uma equação do segundo grau.

A essa altura já aprendemos como criar programas inteiros com as etapas de software que vimos anteriormente: input, processamento e output. Tudo isso em `c#`. Entretanto, no estado atual do código, nossos programas não estão muito competentes em se adaptar a diferentes tipos de inputs ou resultados.

Um exemplo prático é que, nesse segundo programa, podemos ter inputs que retornem um resultado indesejado como podemos ver na imagem abaixo.



```

Console de Depuração do Microsoft Visual Studio
Vamos resolver uma equação do tipo  $ax^2 + bx + c = 0$ 
Qual o valor de a?
1
Qual o valor de b?
2
Qual o valor de c?
3
Dados: a = 1, b = 2 e c = 3
Equação:  $(-(-2) \pm \text{Raiz}((2)^2 - 4*1*3))/2*1 = 0$ 
Resultado: x1 = NaN e x2 = NaN

```

Figura 3.3: Programa de resolução de uma equação do segundo grau com erro.

3.1.5 Estrutura Condicional

A necessidade de lidarmos com diferentes situações ou contextos é o que nos leva a estudarmos as **estruturas condicionais**. Por meio dessas estruturas, podemos construir diferentes outputs de acordo com qualquer lógica que implementarmos e, com isso, evitar vários bugs nos nossos programas.

Operadores Relacionais

Ao lidarmos com estrutura condicionais nós temos alguns operadores que nada mais são do que expressões lógicas que retornam apenas 2 resultados possíveis: true ou false. Esses operadores são como "gatilhos" que são usados para controle do fluxo de processamento do código.

Operador	Operação	Exemplo
==	Igualdade	n1 == n2
<	Menor	a < 10
>	Maior	b > 1
<=	Menor igual	c <= 2
>=	Maior igual	d >= 22
!=	Desigualdade	e != 100

Na tabela acima, temos os operadores relacionais que são usados ao longo das estruturas condicionais simples e compostas.

Tome cuidado com o operador de igualdade == e o de atribuição =, afinal, eles usam o sinal de igualdade mas significam coisas diferentes.

Condição Simples e Composta

Em c# o comando usado para criar uma condição simples é o **if**. Se a condição que colocarmos atrelada ao comando **if** retornar um true, o bloco

de código atribuído a ele será executar, caso contrário, o código do bloco será ignorado.

```
// Parte condicional no calculo da equacao de segundo grau

delta = (Math.Pow(b, 2) - 4 * a * c);

// Output dos resultados com condicao
if (delta >= 0)
{
    Console.WriteLine("Dados: a = {0}, b = {1} e c = {2}", a, b, c);
    Console.WriteLine($"Equação:  $-\{b\} \pm \text{Raiz}(\{b\}^2 - 4\{a\}\{c\})/2\{a\} = 0$ ");
    Console.WriteLine($"Delta: {delta}");
    Console.WriteLine("Resultado: x1 = {0} e x2 = {1}", x1, x2);
};

if (delta < 0)
{
    Console.WriteLine("Dados: a = {0}, b = {1} e c = {2}", a, b, c);
    Console.WriteLine($"Equação:  $-\{b\} \pm \text{Raiz}(\{b\}^2 - 4\{a\}\{c\})/2\{a\} = 0$ ");
    Console.WriteLine($"Delta: {delta}");
    Console.WriteLine("Resultado: X não possui raiz real! ");
};
```

Com essa adaptação, nosso código estará preparado para o caso onde o interior da raiz (que chamamos de delta) seja negativo. Mas podemos ver que o nosso código ficou um pouco esquisito. Criamos dois blocos de código que são, claramente, relacionados entre si: Se um bloco for executado, o outro não será. Para facilitar o trabalho com essas situações, as linguagens de programação possuem as estruturas de condição compostas.

Esses são os casos onde, se o teste lógico retorna true, executamos um bloco, ou o outro bloco será executado caso o resultado lógico seja false. Com isso evitamos ter que criar dois testes com `if`. O token usado em `c#` para uma condição com dois blocos de códigos (ou seja, uma condicional composta) é o `if-else`. Abaixo nós reescrevemos a solução anterior mas agora fazendo uso do condicional composto.

```
// Parte condicional no calculo da equacao de segundo grau

delta = (Math.Pow(b, 2) - 4 * a * c);

// Output dos resultados com condicao
if (delta >= 0)
{
    Console.WriteLine("Dados: a = {0}, b = {1} e c = {2}", a, b, c);
    Console.WriteLine($"Equação:  $-\{b\} \pm \text{Raiz}(\{b\}^2 - 4\{a\}\{c\})/2\{a\} = 0$ ");
    Console.WriteLine($"Delta: {delta}");
    Console.WriteLine("Resultado: x1 = {0} e x2 = {1}", x1, x2);
} else
{
    Console.WriteLine("Dados: a = {0}, b = {1} e c = {2}", a, b, c);
    Console.WriteLine($"Equação:  $-\{b\} \pm \text{Raiz}(\{b\}^2 - 4\{a\}\{c\})/2\{a\} = 0$ ");
    Console.WriteLine($"Delta: {delta}");
};
```

```
Console.WriteLine("Resultado: X não possui raiz real! ");
};
```

Agora sim nosso código está elegante e adaptado para as duas saídas possíveis de resultado para os inputs do usuário.

Operadores Booleanos e Comandos IF Aninhados

Nós aprendemos que o bloco de código será executado sempre que a expressão lógica retornar um resultado true. Também vimos que existem os operadores relacionais que nos ajudam a construir essas expressões. Agora, vamos aprender os operadores booleanos que nada mais são do que os operadores lógicos clássicos (\wedge , \vee , \neg).

Operador	Operação	Exemplo
&& (AND)	true se tudo for true	if (a > 2 && b == 1)
(OR)	true se um for true	if (a < 2 b != 0)
! (NOT)	false se true	if (!(x==y))

Existem problemas que requerem mais de dois resultados no output. Nesse caso, não é suficiente usarmos apenas os tokens de condicional composto if-else da maneira como aprendemos até agora.

Para resolver esse problema temos a técnica de IF aninhados (nested) e a de IF escada (ladder). As duas formas produzem o mesmo resultado mas mudam significativamente legibilidade do nosso código. Abaixo temos um exemplo de cada.

```
// Ladder IF

if (condicao1)
    comando1;
else if (condicao2)
    comando2;
else if (condicao3)
    comando3;
...
else if (condicao_n-1)
    comando_n-1;
else
    comando_n;
```

```
// Nested IF
```



```

if (condicao1)
    comando1;
else
    if (condicao2)
        comando2;
    else
        if (condicao3)
            comando3;
        else
            if (condicao4)
                comando4;
            ...
            else
                if (condicao_n-1)
                    comando_n-1;
                else
                    comando_n;

```

O Comando Switch e o Operador Ternário

Agora que aprendemos a lidar com vários casos condicionais encadeados, podemos construir soluções relativamente complexas com várias saídas diferentes. Entretanto, quando temos muitas situações possíveis, o nosso código pode ficar um pouco ruim de ser lido por outras pessoas.

Pensando nessa necessidade, os criadores do `c#` criaram um operador que simplifica ainda mais nosso trabalho. Sim, é isso mesmo, nós acabamos de aprender IFs aninhados e em escada e já vamos aprender uma maneira melhor de fazer exatamente o que eles fazem. Se acostume com isso. Em tecnologia existem quase sempre várias maneiras de se chegar no mesmo resultado.

O operador que é mais indicado para lidar com várias situações de saída é o `switch-case`. Cujas construção é bem mais legível que os IFs anteriores. Abaixo temos um exemplo.

```

// Lidando com varias condicoes com switch/case

switch(opcao)
{
    case op1:
        comandos1;
        break;
    case op2:

```

```

        comandos2;
        break;
    case op3:
        comandos3;
        break;
    ...
    case op_n:
        comandos_n;
        break;
    default:
        bloco que sera executado se nenhuma
        das anteriores for escolhida;
}

```

Comentário: Existem várias maneiras de se usar o switch. Eu não vou me aprofundar agora em todas elas. Mas vale muito a pena fazer uso dessa ferramenta para cenários de várias interações possíveis ou múltiplas saídas contextuais.

Para finalizar o nosso estudo das estruturas condicionais, vamos aprender como trabalhar com o **Operador ternário**. Esse nome é relativo ao número de operandos que esse comando usa (no caso, 3).

```
condicao ? expressao_true : expressao_false
```

A condição é uma expressão lógica (que usa operadores relacionais e booleanos) que só pode retornar true ou false. A expressão logo após o ponto de interrogação será o retorno do operador em caso de true. A expressão após os dois pontos é o retorno em caso de false.

Sim, é exatamente um caso de if-else só que em apenas uma linha. Como as soluções podem ficar muito grandes, é sempre bom termos em mente que quanto menos linhas, mais fácil será fazer manutenção nos nossos programas. O operador ternário é muito usado por programadores mais experientes. Abaixo temos um exemplo comparando as duas maneiras.

```

// Comparacao entre if-else e operador ternario

bool passou;
double nota;

// if-else
if (nota >= 60)
    passou = true;

```

```

else
    passou = false;

// operador ternario
passou = nota >= 60 ? true : false // maneira 1
passou = (nota >= 60) ? true : false // maneira 2

```

Agora podemos ver claramente a vantagem do uso do operador ternário.

3.1.6 Estrutura de Repetição

Como sabemos, ainda temos mais um tipo de estrutura para estudarmos. Até agora, aprendemos como estruturar um código sequencialmente e como criar blocos de código que só serão executados se determinadas condições previamente definidas forem satisfeitas. Agora, vamos aprender como evitar ter que repetir blocos de códigos.

O c# possui 3 operadores de repetição:

```

// tipo 01
while (condicao)
    comando;

// tipo 02
do
    comando;
while (condicao);

// tipo 03
for (inicial; expressao logica; atualizacao)
    comando;

```

Vamos ver um exemplo de como nosso código pode ser reduzir com o uso da estrutura de repetição. Primeiro, vamos ver um exemplo onde o programa recebe uma lista de 3 palavras (nome, nome do meio e sobrenome) e depois faz o print.

```

// Programa que recebe nome completo e
// depois devolve uma mensagem de boas vindas

string nome, mid, last;

Console.WriteLine(" Seja bem vindo(a)!");

```

```

Console.WriteLine(" Qual seu primeiro nome?");
nome = Console.ReadLine();

Console.WriteLine(" Qual seu nome do meio?");
mid = Console.ReadLine();

Console.WriteLine(" Qual seu sobrenome?");
last = Console.ReadLine();

Console.WriteLine(" Bem Vindo(a)!");

Console.WriteLine(nome);
Console.WriteLine(mid);
Console.WriteLine(last);

Console.WriteLine(" E Vide panel"lte sempre!");

```

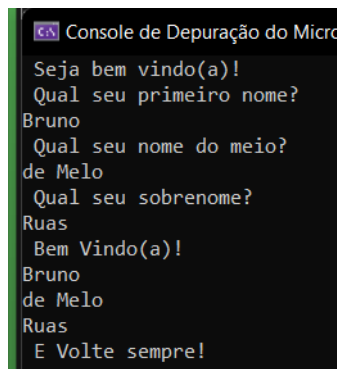


Figura 3.4: Programa simples

Agora vamos refazer esse programa usando os 3 tokens de estrutura de repetição que o `c#` nos dá. Vamos usar a mesma ordem em que elas foram apresentadas.

Os Comandos **WHILE**, **DO WHILE** e **FOR**

Repetição usando **WHILE**

A repetição usando o token `while` é condicionada ao resultado de uma expressão lógica. Caso a expressão lógica retorne `true`, o bloco de código será executado até o seu final. Após a execução, é feita uma nova verificação da expressão lógica. O processo de loop só será finalizado caso a expressão ló-

gica retorne o valor false. Abaixo temos a refatoração⁵ evitando a repetição de partes de partes do código anterior.

```
// Programa de nomes refatorado com while

string nome, mid, last;
int contador;

// nome default
nome = "";
mid = "";
last = "";

Console.WriteLine(" Seja bem vindo(a)!");

Console.WriteLine(" Por favor, insira o seu nome completo.");
Console.WriteLine(" Obs. No máximo 3 palavras");
contador = 1;

while (contador <= 3)
{
    Console.WriteLine($" Palavra nº {contador} do seu nome?");
    if (contador == 1)
        nome = Console.ReadLine();
    else if (contador == 2)
        mid = Console.ReadLine();
    else if (contador == 3)
        last = Console.ReadLine();

    contador += 1;
};

Console.WriteLine(" Bem Vindo(a)!");
Console.WriteLine(nome);
Console.WriteLine(mid);
Console.WriteLine(last);
Console.WriteLine(" E Volte sempre!");
```

Agora nós temos um loop usando a expressão "a variável contador é menor igual a 3?" sempre que a resposta for sim, o programa perguntará qual a n-ésima palavra do nome da pessoa. Pode parecer que nesse exemplo o uso

⁵Refatorar é o processo de mudar o código e obter o mesmo resultado no final. É uma ótima prática a ser feita.

do loop mais complicou do que facilitou. Mas o objetivo aqui é exercitar o uso desse operador, então devemos focar em compreender como usar o while.

Repetição usando DO-WHILE

Para a nossa sorte, o operador DO WHILE é muito similar ao WHILE. A única diferença é que primeiro declaramos o bloco de código e, no final, colocamos a condição para sua execução. No exemplo abaixo temos o mesmo resultado que o de cima mas usando essa outra estrutura de operador.

```
// Programa de nomes refatorado com do-while

string nome, mid, last;
int contador;

// nome default
nome = "";
mid = "";
last = "";

Console.WriteLine(" Seja bem vindo(a)!");

Console.WriteLine(" Por favor, insira o seu nome completo.");
Console.WriteLine(" Obs. No máximo 3 palavras");
contador = 1;

do
{
    Console.WriteLine($" Palavra nº {contador} do seu nome?");
    if (contador == 1)
        nome = Console.ReadLine();
    else if (contador == 2)
        mid = Console.ReadLine();
    else if (contador == 3)
        last = Console.ReadLine();

    contador += 1;
} while (contador <= 3);

Console.WriteLine(" Bem Vindo(a)!");
Console.WriteLine(nome);
Console.WriteLine(mid);
Console.WriteLine(last);
```

```
Console.WriteLine(" E Volte sempre!");
```

Repetição usando FOR

Quando usamos nosso operador `while`, foi criada uma variável `contador` que era incrementada a cada iteração até que o teste lógico "menor igual a 3" retorne `false`. A vantagem do operador de loop `for` é que podemos fazer isso diretamente no parâmetro da função.

O operador de loop `for` recebe 3 parâmetros, na ordem: condição inicial da variável de controle; expressão lógica (que retorna `true` ou `false`) e, por fim, um incremento ou decremento.

```
// Programa de nomes refatorado com for

string nome, mid, last;
int contador;

// nome default
nome = "";
mid = "";
last = "";

Console.WriteLine(" Seja bem vindo(a)!");

Console.WriteLine(" Por favor, insira o seu nome completo.");
Console.WriteLine(" Obs. No máximo 3 palavras");
contador = 1;

for (contador = 1; contador <= 3; contador++)
{
    Console.WriteLine($" Qual a palavra nº {contador} do seu nome?");
    if (contador == 1)
        nome = Console.ReadLine();
    else if (contador == 2)
        mid = Console.ReadLine();
    else if (contador == 3)
        last = Console.ReadLine();
};

Console.WriteLine(" Bem Vindo(a)!");
Console.WriteLine(nome);
Console.WriteLine(mid);
Console.WriteLine(last);
Console.WriteLine(" E Volte sempre!");
```

Contadores e Acumuladores

Agora sabemos reduzir nossos códigos usando 3 tipos de tokens fornecidos pelo `c#`. Vamos ver um pouquinho mais a respeito do atributo de contador do loop `for`.

Um **contador** nada mais é do que uma variável do tipo constante que

receberá a atualização do seu valor a cada loop. Se a situação inicial do nosso contador for igual a 1, a cada iteração ele receberá a expressão `contador = contador + 1`.

Podemos criar um contador diretamente no código através da atribuição do valor pela soma. Mas o token de loop `for` recebe um terceiro atributo que é justamente feito para criarmos nossa condição de atualização. Quando colocamos o terceiro parâmetro igual a `contador++`, a cada iteração teremos `+1` atribuído ao contador.

Um **acumulador** é muito parecido com um contador, a diferença é que a cada iteração nós podemos adicionar qualquer valor ao acumulador. Por exemplo, se queremos calcular a média de uma turma, teremos que somar todas as notas e dividir pelo quantitativo dos alunos da turma. Para isso, podemos criar um acumulador chamado `soma_notas` que recebe, para cada aluno, a nota através de um comando parecido com o exemplo a baixo.

```
double soma_notas = 0;

for (n_aluno=0; n_alunos <= qtd; n_aluno++)
{
    soma_notas = soma_notas + nota
};
```

A cada iteração, somamos o valor da variável `nota` à variável `soma_notas`.

3.2 Manipulação de Dados em Memória Primária e Secundária

Ao longo do material nós usamos várias vezes frases como "a medida que nosso código cresce" ou "para manter a simplicidade no processo de atualização" e outras frases de mesmo teor. Isso não é em vão. Quando olhamos os códigos dos sistemas operacionais, por exemplo, podemos ver que a quantidade de linhas de código passam da casa do milhão. Manter tudo isso funcionando de modo coeso e atualizável é uma tarefa que demanda muita habilidade e planejamento.

Para nos ajudar nesse trabalho, vamos aprender sobre métodos de organização de código através do uso de funções e procedimentos.

Comentário: Cada linguagem podem ter nomes diferentes para esses conceitos, mas o importante é aprender a lógica da coisa.

3.2.1 Criando e Usando Funções e Procedimentos

Em linguagens orientadas à objeto como o `c#`, podemos usar os objetos para criação de rotinas que generalizam atividades de modo a reduzir o volume de linhas usadas no nosso código. A ideia é simples: menos linhas, mais fácil será a manutenção.

Normalmente, quando estamos desenvolvendo uma solução de software, primeiro nós criamos um algoritmo que descreve nos pormenores o que será feito e como será feito. Essa primeira etapa é focada na solução do problema.

A partir dela, devemos pensar em encontrar procedimentos que se repetem ao longo do código ou que podem ser usados para outros trabalhos similares. Sempre que identificarmos padrões que podem ser reutilizados, estamos diante de uma melhoria por modularização.

Existem dois tipos de modularização que podemos usar: as que retornar um valor (que chamamos de funções) e as que não retornam valores⁶ (que chamamos de procedimentos). Abaixo vemos como criar esses dois tipos.

```
// criando um metodo/funcao que retorna um inteiro

int nome_metodo(string par1, float par2,..., int parn)
{
    int valor_resultado;

    comando1;
    ...

    return valor_retorno;
};

// criando um procedimento
static void nome_procedimento (string par1, double par2,..., int parn)
{
    comando1;
    ...
};
```

Temos que lembrar que um procedimento em `c#` sempre vai receber esse token `void` antes de ser criado (ainda não aprenderemos o que significa esse termo `static`). Lembremos, também, que um **método** nada mais é que uma função que será atribuída dentro de um objeto e que temos que definir o tipo da variável de retorno na hora da criação dele.

Para usarmos um parâmetro não precisamos fazer nenhuma atribuição,

⁶Isso não significa que não podemos printar informação na tela por meio de um procedimento, só significa que o resultado dele não salvará uma variável na memória do computador, logo, não poderá ser atribuída a uma nova variável.

basta chamar-lo com os parâmetros (par1 até parn) preenchidos. No caso de uma função, como ela retorna um resultado, precisamos atribuir o resultado a alguma variável.

```
// chamando um procedimento  
nome_procedimento(par1,par2);  
  
// chamando um metodo  
resultado = nome_metodo(par1,par2);
```

Comentário: Se não queremos usar nenhum parâmetro, basta definir o procedimento ou o método com os parênteses vazios, por exemplo, `nome_procedimento()`.

3.2.2 Passagem de Parâmetros

3.2.3 Manipulação de Arquivos em C#

Capítulo 4

Desenvolvimento Web Front-End

Bibliografia

Bibliografia Básica

- SIKOS, L. Web Standards. Mastering HTML5, CSS3, and XML.
- DACONTA, M. C.; SMITH, K. T.; OBRST, L. J. The semantic Web: a guide to the future of XML, Web services, and knowledge management. [s. l.]: Wiley, [s. d.]. ISBN 0471432571
- SILVA, Maurício Samy. HTML 5: a linguagem de marcação quer revolucionar a web. 2. ed. rev. e ampl. [s. l.]: Novatec, 2014. ISBN 9788575224038
- SANDERS, William B. Smashing HTML5: técnicas para a nova geração da web. Porto Alegre: Bookman, 2012. xiv, 354 p. ISBN 9788577809608
- DEITEL, Paul J., Deitel, Harvey M. Ajax, Rich. Internet Applications e Desenvolvimento Web para Programadores. Pearson 776. ISBN 9788576051619
- SILVA, Maurício Samy. CSS3: desenvolva aplicações web profissionais com uso dos poderosos recursos de estilização das CSS3. São Paulo: Novatec, 2011. 494 p. ISBN 9788575222898
- BERTAGNOLLI, S. de C.; MILETTO, E. M. Desenvolvimento de software II: introdução ao desenvolvimento web com HTML, CSS, JavaScript e PHP. [s. l.]: Bookman, 2014. ISBN 9788582601952

4.1 A Web: Evolução, Padrões e Arquitetura

4.1.1 Histórico e Evolução da Web

A Web é um sistema da informação de hipertextos onde o acesso é feito por meio de **navegadores (browsers)**.

Existem alguns protocolos comuns para transferência de alguns tipos de arquivos. Para mensagens (e-mail) usamos o **SMTP**, para transferência de arquivos usamos o **FTP**, aplicações de telefonia usam o **VOIP** e para páginas de conteúdo usamos o **HTML**.

A história da web eu ainda vou colocar aqui quando tiver mais tempo.

4.1.2 W3C e os Padrões da Web

O WORLD WIDE WEB CONSORTIUM (W3C) é uma organização sem fins lucrativos cujo líder é o Tim Berners-Lee, justamente o inventor da Web. Existem várias organizações ao longo do planeta que fazem parte desse consórcio internacional.

O W3C mantém a gestão de vários padrões usados todos os dias:

- Design e Aplicações Web (HTML, CSS, SVG, Ajax, Acessibilidade);
- Arquitetura da Web (Protocolo HTTP, URI);
- Web Semântica (Linked Data - RDF, OWL, SPARQL);
- Web Services (SOAP, WSDL);
- Tecnologia XML (XML, XML Schema, XSLT);
- Navegadores e ferramentas de autoria.

A W3C possui um processo de publicação das normativas. Normalmente, o fluxo é:

- Working Draft (WD)
- Candidate Release (CR)
- Proposed Recommendation (PR)
- Recommendation (REC)

4.1.3 Componentes da Arquitetura da Web

A web pode ser entendida como uma coleção de componentes que permitem a comunicação entre o cliente e os servidores de aplicações. Os principais componentes dessa arquitetura são:

- Ambiente Cliente (Client Web)

Geralmente um Browser que envia as requisições usando o protocolo HTTP(S) para o servidor web através de uma rede de computadores.

- Ambiente Servidor

O ambiente servidor possui vários componentes (banco de dados, aplicações, API e etc) mas o principal componente é o servidor web. Ele recebe a requisição HTTP(S) do client, interpreta a URL e envia os recursos solicitados (HTML, CSS, JS, JPEG, MP4 e etc) por meio da rede.

- Internet

É a rede mundial baseada no protocolo TCP/IP onde todo computador conectado é denominado host (hospedeiro) e possui um identificador de endereço IP (internet protocol) que possui determinados padrões.

- URI (uniform resource locator)

Como o nome indica, um URI é um localizado que pode ser classificado em duas maneiras. O URL é o tipo de URI que usa o endereço do conteúdo como método de localização, ele nos diz onde encontrar o recurso (por exemplo, o caminho `c://home/desktop/test.txt`). O URN é o tipo que usa o nome do recurso, ele nos diz a identidade do item procurado (por exemplo, o sistema ISBN).

- Requisição

É o pacote de dados enviado pelo client através da internet para o web server onde está a instrução do que deve ser enviado como resposta.

- Resposta

Como o nome já diz, é o retorno do web server ao client com os dados requisitados.

- Protocolo HTTP

É o padrão como client e web server se comunicam pela rede.

4.1.4 URI, URL e URN

Já vimos que o URI abarca dos conceitos de URL e URN. Agora vamos aprender um pouco mais sobre os padrões de endereços em ambos os protocolos.

URL

O padrão URL serve para identificar o recurso pela sua localização e é composto da seguinte maneira:

```
ftp://example.com:8080/pasta/arquivo?name=book#nose
      Cujas partes são
scheme://authority/path?query#fragment
```

Como podemos ver, a URL é composta por várias partes:

- scheme - é a forma de interação (ftp, http, https, ...).
- user:pass - são as informações do user.
- host - endereço de ip do server.
- porta - qual a porta TCP/IP do server (o padrão http é 80 e pode ser omitida).
- path - local onde o recurso se encontra.
- query - detalhe da consulta na forma de pares nome-valor.
- fragmento - qual seção do recurso.

URN

```
urn:example:animal:ferret:nose
      Cujas partes são
scheme:path:authority
```

A URN apenas nos dá um recurso específico (NSS) contido em algum namespace (NID) sem qualquer informação sobre onde o arquivo está localizado.

4.1.5 Protocolo HTTP

O hypertext transfer protocol é mantido pela W3C e rege a camada de aplicação dos sistemas distribuídos de informação em hipertexto. Existem muitas versões mas a mais utilizada é a 2.0 de 2015.

Para entender melhor o http, consideremos o processo usual de navegação na web:

1. user informa a URL
2. client monta a requisição http e envia ao web server
3. server recebe a requisição e envia a resposta ao client
4. a resposta é recebida e interpretada pelo browser com os dados exibidos ao user
5. dependendo da página, pode ser que novas requisições sejam feitas para que todos os componentes sejam carregados propriamente.

Podemos ver que o http é o conjunto de regras que rege a comunicação client-server da web.

Partes da requisição HTTP

Uma requisição é formada por 3 partes:

- Linha de Requisição
 - Método
 - * GET - Requisita dados.
 - * POST - Envia dados para o server.
 - * HEAD - Requisita dados mas o retorno deve ser um conjunto de cabeçalhos.
 - * PUT - Criação ou Atualização de dados.
 - * DELETE - Excluir algum dado.
 - * TRACE - Solicita uma cópia da requisição (serve pra testar integridade).
 - * PATCH - Alterações parciais em um recurso.
 - * OPTIONS - Lista de métodos e opções disponíveis para o server.
 - * CONNECT - Usado quando o client se conecta com o server via proxy.
 - Recurso - É o caminho do dado requerido.
 - Versão do Protocolo - Qual versão do http será usada.
- Linhas de Cabeçalho
 - Inclui informações complementares à requisição sendo formado por pares nome-valor.
- Corpo da Entidade
 - Dados adicionais como forms, arquivos para upload e etc.

Partes da resposta HTTP

- Linha de Resposta
 - Versão HTTP.
 - Código de Retorno.
 - Mensagem explicativa do código de retorno.
- Linhas de Cabeçalho

Uma informação importante que consta no cabeçalho é o `content-type`. Ele diz o formato do conteúdo enviado como resposta¹. Essa informação é apresentada conforme os MIME Types.
- Corpo da Entidade É o recurso solicitado pelo client (html, css, js, jpeg, mp4, ...)

Um pouco mais sobre Cabeçalhos

Os cabeçalhos possuem características parecidas tanto nas requests quanto nas responses. Podemos classifica-los como sendo dos tipos:

- Request header - Informações sobre o client ou a requisição feita.
- Response header - Informações sobre a resposta ou sobre o web server.
- Entity header - Informações sobre o conteúdo da entidade trocada (tamanho e tipo).
- General header - Informações gerais usadas tanto em requests quanto em responses.

4.1.6 Servidores Web

Você só consegue visualizar as informações de um site porque o servidor web foi capaz de interpretar a requisição feita pelo seu browser e responder com o conteúdo adequado. Agora vamos entender um pouco mais sobre o ambiente servidor.

O principal elemento do ambiente servidor é o web server. Ele é quem recebe, interpreta e responde as requisições dos clients ao longo da internet. Podemos também incluir outros elementos importantes no ambiente servidor como o **servidor de banco de dados** e os **servidores de serviços (APIs)**.

¹ Isso pode ser a causa de alguns bugs na sua aplicação.

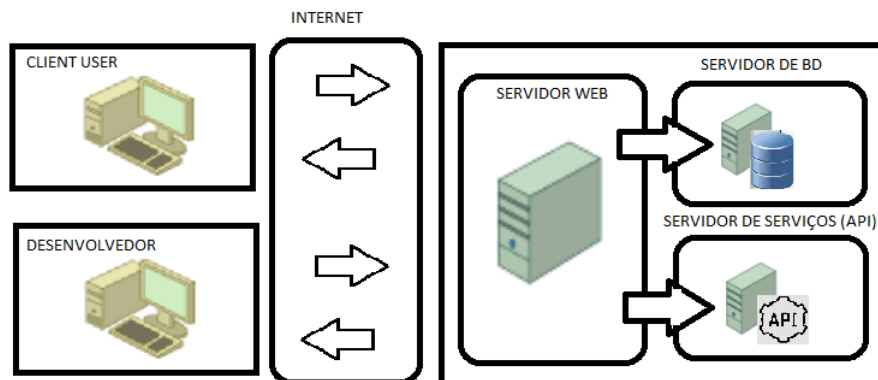


Figura 4.1: Esboço da arquitetura web

Funções de um web service

Um web server nada mais é que um software rodando em uma máquina. Ele desempenha várias funções que podemos elencar como:

- Atender as requests http e responder a elas.
- Gerenciar sites.
- Gerenciar arquivos dos sites.
- Integrar mecanismos de scripts: php, perl, aspx, Ruby, Python e etc.
- Autenticar users (básica ou com servidores de autenticação).
- Implementar criptografia nas comunicações (https - tls/ssl).
- Cache de recursos.
- Auditoria das alterações e logs.

Software e Provedores

Basicamente, existem 3 formas de tornar uma aplicação web acessível aos clients: Rodar um web server na máquina local; instalar e configurar um web server em uma máquina dedicada para esse trabalho e, por fim, contratar um provedor que ofereça esse serviço.

A lista de softwares que se propõe a fazer o trabalho de um web server é enorme. O material do curso elenca dois:

- Apache HTTP Server | Apache Web Server
É um open source multi plataforma. Permite execução de multilinguagens como php, perl entre outras. Uma maneira simples de instalar

é pelo XAMPP (que já integra o apache web server, banco de dados MariaDB e um ambiente PHP e Perl).

- Microsoft Internet Information Server (IIS)
É a solução proprietária da Microsoft. Baseado na plataforma .NET, permite hospedar sites estáticos. O IIS já vem disponível junto dos SO Windows.

A lista de provedores também é extensa e possuem diferentes capacidades distintas mas podemos destacar algumas ferramentas úteis:

- Servidores em Nuvem
 - Azure
 - Heroku²
 - AWS
- Editores e IDEs online
 - Replit
 - CodeSandbox
 - Glitch
 - GitHub Pages

4.1.7 Dinâmica de Aplicações Web

Quando você acessa um site, o arquivo que coordena o modo de exposição da informação e os conteúdos da mesma é um arquivo “.html”. Observe o exemplo abaixo de uma página simples.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Document</title>
    <link rel="stylesheet" href="style.css">
    <script> src='app.js'</script>
  </head>
  <body>
    <img src='logo.jpg' alt="imagem_logo">
  </body>
</html>
```

²Esse aqui eu to estudando e fazendo um manual de como usar. Você pode ler o manual nesse [LINK]

As tags que contém as partes `style.css`, `app.js` e `logo.jpg` fazem menção à outros arquivos que farão parte da composição da página. Alguns são referentes à funcionalidades ou layout da aplicação enquanto outros podem ser referentes à conteúdos mostrados na página.

Uma vez que o servidor compreende a request feita pelo client, ele envia uma série de arquivos que serão lidos pelo browser do usuário e serão interpretados por ele. O html é justamente o primeiro arquivo lido porque ele diz ao navegador quais conteúdos mostrar e, a partir das referências contidas no html, como mostrar e quais funcionalidades a página terá.

O processamento de um site

1. O client envia uma requisição via http (com o método GET) para o web server
2. O server envia o arquivo html da página requisitada para o browser
3. Ao processar o html, o browser percebe que ele faz menção de outros arquivos (como css, js, mp3, etc)
4. O browser faz novas requisições ao server até ter todos os arquivos necessários para o carregamento da página

Como você pode ver, é muita coisa acontecendo. Só não nos damos conta disso porque o processo é muito rápido hoje em dia devida a velocidade das nossas conexões banda larga. Lembrando sempre que todas as requisições e respostas entre client e server são feitas usando-se o protocolo HTTP que a gente viu logo antes.

4.2 Desenvolvimento de Interfaces Web

4.2.1 A Linguagem HTML

A linguagem HTML foi criada por Tim Berners-Lee no ano de 1991 e foi baseada no padrão Standard Generalized Markup Language (SGML). Seu escopo original era para permitir a divulgação de pesquisas científicas.

Com o passar dos anos, novas tecnologias foram somadas ao ecossistema para facilitar o processo de construção das soluções web. O Cascading Style Sheet (CSS) foi criado para facilitar o desenvolvimento do conteúdo separando a parte de estilo e aparência do conteúdo em HTML. O JavaScript permitiu a manipulação de elementos além de dar mais dinâmica para as páginas web.

O W3C foi criado em 1993 e, a partir dessa data, o HTML foi mantido e padronizado por essa organização. Desde então a linguagem vem sendo alterada para permitir sua evolução.

Em 2004 foi criado o Web Hypertext Application Technology Working Group (WHATWG) por pessoas da Apple, Mozilla e Opera. Na época, o W3C estava trabalhando no padrão XHTML 2.0 (que iria substituir o HTML 4.01) mas o WHATWG conseguiu propor um monstro que acabou sendo o HTML 5. O HTML 5 foi recebido e amplamente adotado no desenvolvimento de aplicações hoje em dia.

Panorama de uma Aplicação

Nós já sabemos que um client faz uma requisição ao web server por HTTP e esse, por sua vez, responde a requisição com, normalmente, um arquivo HTML. De posse de arquivo, o browser consegue saber se precisará solicitar mais arquivos ao web server até que todas as referências do HTML sejam satisfeitas e a página carregada.

A grosso modo, podemos dizer que o HTML pode fazer menções a arquivos dos seguintes tipos:

- CSS
- Arquivos de Multimídia
- JavaScript
- RIA - Rich Internet Applications
 - Applet Java
 - Adobe Flash
 - Adobe Air
 - Adobe Flex
 - SilverLight

Se o site utiliza soluções dinâmicas como PHP, Java, Python, Ruby ou ASP.NET, quando a requisição é feita, o web server primeiro faz o processamento desses arquivos (normalmente por um outro servidor de APIs) e o resultado serão outros arquivos HTML, CSS, JS ou Multimídia. Após o processamento, o resultado é enviado para o client que será atualizado pelo browser.

Nas aplicações modernas, o seu browser está em processo praticamente contínuo de interação com o servidor e vice-versa.

A Sintaxe da Linguagem HTML

Uma página HTML é uma coleção de **elementos**. Você consegue identificá-los facilmente porque estão entre os pares de símbolos <>. Cada elemento também tem uma tag de abertura e uma de fechamento. Por exemplo:

```
<body> Aqui vai o conteúdo do body </body>
```

Também existem elementos que não precisam do par de tags de abertura e fechamento. Por exemplo:

```
<input disable name='Nome' value='rommelcarneiro'>
```

Atente para o fato que alguns elementos aceitam outros elementos internamente. Por exemplo, dentro do elemento <body></body> nós colocamos todos os outros elementos que comporão a nossa página web, como por exemplo, formulários, parágrafos, vídeos e etc. Então se acostume de termos elementos dentro de outros elementos.

Dentro de alguns elementos podem ser inseridas informações e configurações por meio de parâmetros que chamamos de **atributos** do elemento. Por exemplo, no elemento logo acima, temos os atributos **name** e **value**.

Agora que sabemos o que são elementos e como eles são construídos, podemos seguir para a **organização de um documento HTML**. Existe um padrão em todo arquivo HTML onde existem alguns elementos obrigatórios para o processamento da página pelo browser do client.

<!DOCTYPE html>	----->	Elemento da versão do HTML
<html lang="en">	----->	Abertura do documento HTML
<head>	----->	Abertura do cabeçalho
<meta charset="UTF-8">	-->	Atributo nome = "valor"
<title>Document</title>	->	Elemento de Título
</head>	----->	Fechamento do cabeçalho
<body>	----->	Abertura do corpo
	----	Elemento de imagem
</body>	----->	Fechamento do corpo
</html>	----->	Fechamento do HTML

Preâmbulo

Como podemos ver, primeiro temos o preâmbulo DOCTYPE, seguido do <html> </html> onde temos outros dois elementos maiores, o cabeçalho (<head> </head>)

e o corpo (`<body> </body>`).

O preâmbulo diz ao navegador qual versão da HTML será usada. Se ele não for indicado, o navegador vai tentar “adivinhar” qual a melhor maneira de interpretar a sua página (chamamos isso de **quirks mode**). Caso você informe qual a versão, o browser usará o processamento adequado (chamamos de **strict mode**). Os formatos do preâmbulo mudam de acordo com a versão do HTML:

- HTML 5
`<!DOCTYPE html>`
- HTML 4.01
`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">`
- HTML 1.0
`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">`

Cabeçalho

É a primeira parte dentro da tag de html. Nele estão as informações sobre o documento de modo a organizar as referências de funcionalidade que serão usadas para o processamento da página web. Podemos resumir os elementos no cabeçalho como:

- title - `<title> </title>`
Define o título do documento. Que também afeta a aba do navegador.
- link - `<link rel="relacao" href="link_do_arquivo.extensao">`
Define as ligações externas como arquivos, scripts, CSS e etc.
- style - `<link rel="stylesheet" href="style.css">`
É um tipo de link. Nele é que vamos indicar qual o arquivo que regerá o layout da nossa aplicação.
- meta - `meta name="nome" content="conteudo">`
Aqui teremos as informações adicionais sobre a página: codificação de caracteres, descrição, palavras-chaves, autor e etc.

Corpo

A segunda parte do html é o corpo. Nele é onde colocamos o conteúdo que fará parte da página. Como é muito comum de se encontrar nos sites, esse conteúdo pode vir mesclado em várias mídias como texto, imagens, vídeos, mapas e etc. Veremos com calma um pouco mais a frente.

Elementos de Texto e Multimídia

Como esse material tem o objetivo de ser para futuras consultas. Eu vou colocar as tags com um pequeno resumo mas não vou comentar muito sobre elas.

Parágrafos e Títulos

Elemento	Tags
Títulos	<code><h1></h1>, ... ,<h6></h6></code>
Parágrafo	<code><p></p></code>
Quebra de Linha	<code>
</code>
Itálico	<code><i></i></code>
Negrito	<code></code>
Importância	<code></code>
Código-fonte	<code><code></code></code>
Texto pre-formatado	<code><pre></pre></code>
Citações	<code><blockquote></blockquote></code>

Enquanto estamos montando a nossa página html, devemos evitar usar os elementos dela para a formatação de layout da nossa solução. É altamente recomendado deixar toda essa responsabilidade para a nossa Cascading Style Sheets (CSS) e focar apenas no conteúdo textual da página web.

Listas

Existem 3 tipos de listas em HTML.

Listas ordenadas:

```
<ol>
  <li> Primeiro item </li> -----> 1. Primeiro item
  <li> Segundo item </li> -----> 2. Segundo item
  <li> Terceiro item </li> -----> 3. Terceiro item
</ol>
```

Lista não ordenada:

```
<ul>
  <li> Primeiro item </li> -----> o Primeiro item
  <li> Segundo item </li> -----> o Segundo item
  <li> Terceiro item </li> -----> o Terceiro item
</ul>
```

Lista de definições:

```

<dl>
  <dt> Termo 01 </li> -----> Termo 01
  <dd> Definição 01 </li> -----> Definição 01
  <dt> Termo 02 </li> -----> Termo 02
  <dd> Definição 02 </li> -----> Definição 02
</dl>

```

Imagens

```

```

Links

```

<a href="link.com" target="_blank"> Texto </a> -----> Nova tab
<a href="link.com" target="_self"> Texto </a> -----> Mesma tab
<a href="link.com" target="_parent"> Texto </a> -----> Frame pai
<a href="link.com" target="_top"> Texto </a> -----> Janela atual
<a href="link.com" target="nome_frame"> Texto </a> --> Frame nominado

```

Elementos Estruturais

A partir da versão 4.0 o principal elemento usado para segmentar as partes de uma página html passou a ser o `<div>` que é um elemento de divisão genérico para agrupar qualquer conjunto de elementos necessários. Por exemplo:

```

<div>
  <h1> Titulo </h1>
  <p> Parágrafo pequeno </p>
  <ol>
    <li>Item</li>
    <li>Item</li>
  </ol>
</div>

```

Na versão 5 do HTML passamos a ter vários tipos de elementos com a mesma função dos `<div>` mas agora com nomes mais fáceis de usar. As vezes nos referimos a eles como **elementos semânticos**. Os novos elementos semânticos apresentados na versão 5 do html são:

Elementos	Descrição
<code><article></code>	Define um artigo
<code><aside></code>	Conteúdo ao lado da página
<code><details></code>	Detalhes adicionais
<code><figcaption></code>	Título para <code><figure></code>
<code><figure></code>	Elemento autocontido
<code><footer></code>	Rodapé para seção
<code><header></code>	Cabeçalho para seção
<code><main></code>	Conteúdo principal
<code><mark></code>	Texto destacado
<code><nav></code>	Conteúdo de navegação
<code><section></code>	Seção do documento
<code><summary></code>	Resumo
<code><time></code>	Define data/hora

Quando construímos a estrutura do nosso site apenas com elementos `<div>` genéricos, nós não estamos indicando nenhuma relação entre essas seções. Quando usamos a divisão via elementos semânticos, permitimos um processamento por algoritmos de modo a abrir todo um leque de possibilidades de interações a partir disso. Esse é um dos motivos que justificam o nome da web 3.0 como sendo **web semântica**.

Abaixo temos duas maneiras de representar uma estrutura de um site. A primeira em estrutura genérica de `div` e a outra em elementos semânticos. Veja como a segunda abordagem é mais simples de ler.

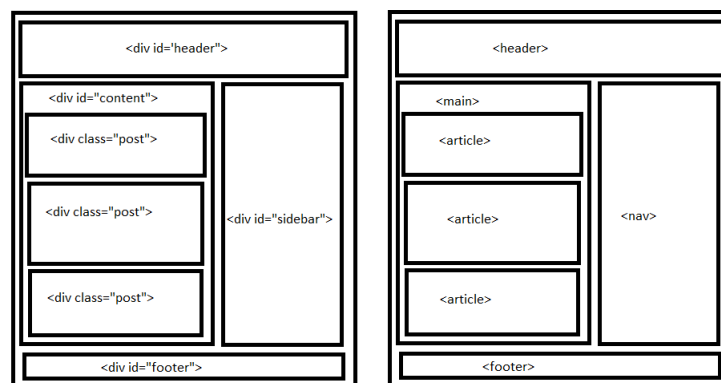


Figura 4.2: Estrutura em `<div>` versus Estrutura em elementos semânticos

Não é difícil perceber que o uso de elementos semânticos é fortemente indicado para o desenvolvimento de aplicações web modernas.

Elementos de Tabelas

Não é nada incomum ter que demonstrar dados usando uma tabela. Pensando nisso, a linguagem HTML também possui um elemento especificamente criado para criação de tabelas. Uma tabela pode ser criada com o uso das seguintes tags:

```
<table border="1"> -----> Cria a Tabela
  <caption> Título </caption> -> Coloca um Título
  <tr> -----> Table Row (tr)
    <td>L1C1</td> -----> Table Data Column 1
    <td>L1C2</td> -----> Table Data Column 2
  </tr>
  <tr>
    <td>L2C1</td> -----> Table Data Column 1
    <td>L2C2</td> -----> Table Data Column 2
  </tr>
</table>
```

Existem vários elementos que podem ser usados dentro de uma tabela. São os principais:

Elementos	Descrição
<table>	Elemento que cria a tabela
<caption>	Título da tabela
<thead>	Linhas do cabeçalho
<tbody>	Linhas do body
<tfoot>	Linhas do rodapé
<tr>	Linha da tabela
<th>	Cabeçalho dentro de uma linha
<td>	Table data

Comentário: Não podemos cair na tentação de usar tabelas como ferramenta de layout da página. Pode até parecer mais simples no começo mas tabelas não são boas para criação de aplicações fluidas e dinâmicas.

Elementos de Formulários

Uma das interações mais básicas que precisamos de um usuário é a inserção de dados na aplicação. Dentre as várias maneiras de conseguirmos um dado inserido pelo usuário, o formulário é a mais simples.

O HTML fornece vários atributos dentro do elemento <form></form> que nos permite a criar campos de texto, botões clicáveis, campos de senha e etc. A sintaxe mais básica de um formulário é dada por:

```
<form name="form_name" action="login.html" method="POST">
  Usuário: <br>
  <input type="text" name="user" value=""> <br>
  Senha: <br>
  <input type="password" name="psw" value=""> <br> <br>
  <input type="submit" value="OK">
</form>
```

Podemos usar o atributo **name** ou **id** para identificar o nosso formulário³. O atributo **action** indica qual URL vai ser disparada uma vez processado o form (no nosso exemplo seria algo como `http://server.com/login.html`). O atributo **method** indica o método HTTP de submissão dos dados do formulário no nosso bando de dados (pode ser **POST** ou **GET**).

Quando o método usado for o **GET**, o browser faz uma requisição da URL indicada para o servidor passando os parâmetros de input como **querystring** na URL. No nosso exemplo, ficaria como `http://server.com/login.html/login.html?user=texto&psw=123`.

Quando o método escolhido é o **POST**, os dados são enviados ao servidor no corpo da requisição HTTP e não aparecem na URL. A essa altura você já deve ser capaz de entender as diferenças entre esses dois métodos.

Elemento <input>

Esse elemento é bastante utilizado na composição dos formulários (na verdade, eu nem consigo pensar em um formulário sem pelo menos um input). Ele define os campos ou entradas de informação e possui os seguintes atributos:

- **type** - Cada tipo de input possui uma visualização diferente quando a página é carregada. Isso é feito para permitir uma melhor interação do usuário de acordo com a natureza da informação requerida. As opções são:
 - **text** - Campo de texto aberto. A quantidade de caracteres pode ser controlada pelo atributo **maxlength**.
 - **number** - Só aceita número como input e permite a seleção por umas setinhas que aparecem ao lado do campo.
 - **password** - Igual ao campo texto mas com os caracteres anonimizados.

³Isso é muito importante porque vamos usar essa informação para fazer alguma coisa.

- email - Confere se o texto inserido possui um @ antes de salvar o formulário.
 - date - Coloca uma máscara no formato de data e cria uma opção de input por calendário.
 - radio button - Uma opção clicável com um valor associado e um nome. O navegador só permite que um único radio button esteja selecionado se existir mais de uma opção com o mesmo nome no atributo **name**.
 - checkbox - Mesma lógica do radio button mas com permissão de vários selecionados simultaneamente.
 - submit - É um botão clicável que normalmente dispara a informação do formulário ao servidor web ou a um script JS local.
 - reset - É igual um submit mas a única função dele é apagar tudo que foi preenchido no formulário.
- **name** - Nome de identificação do campo.
 - **value** - Valor contido no campo.
 - **placeholder** - Valor que aparece quando o campo estiver vazio.
 - **required** - Validação automática para evitar o não preenchimento do campo antes da submissão do form.
 - **disabled** - Inativa o campo e não permite interação mas o user ainda poderá ver.

Na imagem abaixo podemos ver como cada tipo do elemento `<input>` aparece para um usuário:

The image shows a collection of HTML form elements. At the top, there are labels for 'Texto:', 'Número:', 'Password:', 'Email:', 'Date:', 'Radio:', 'Checkbox:', 'Submit:', and 'Reset:'. Each label is followed by its corresponding input field. The 'Texto:' field contains 'textotext'. The 'Número:' field contains '123456'. The 'Password:' field is filled with dots. The 'Email:' field contains 'teste@teste.com'. The 'Date:' field contains '10/11/1111' and has a small calendar icon to its right. The 'Radio:' section shows two radio buttons, with the first one selected. The 'Checkbox:' section shows a single checkbox that is checked. The 'Submit:' and 'Reset:' labels are followed by buttons labeled 'Enviar' and 'Redefinir' respectively.

Figura 4.3: Tipos de elementos `<input>` dentro de um formulário html.

Elemento `<textarea>`

Esse é tranquilo de entender. Sempre que precisarmos de um input de texto maior do que uma linha, podemos usar o elemento `<textarea name="" rows="10" cols="50"></textarea>` para isso. É possível alterar a quantidade de linhas e a número de colunas para apresentação da nossa caixa de texto apenas mudando os parâmetros dos atributos.

Elemento `<select>`

Podemos permitir que o usuário selecione uma lista pré-selecionada de opções através de uma **lista em caixa** (também chamada de **dropdown menu**). Um exemplo de código contendo esse elemento por ser visto abaixo.

```
<label for="lista"> Dropdown Menu </label>
<select name="lista">
  <option value="">Selecione uma opção</option>
  <option value="01">Opção 01</option>
  <option value="02">Opção 02</option>
  <option value="03">Opção 03</option>
  <option value="04">Opção 04</option>
  <option value="05">Opção 05</option>
</select>
```

É possível transformar a lista suspensa em uma lista fixa que permite mais de uma seleção. Para fazer isso é só adicionar o atributo `multiple` e também o atributo `size=` no elemento `select`.

Perceba que além do elemento de lista nós trouxemos um novo elemento chamado `label` que adiciona um texto associado a algum elemento. No nosso exemplo, veja como foi indicado no atributo `for` o mesmo nome que o atributo `name` recebe dentro do elemento `select`.

O resultado pode ser visto abaixo:

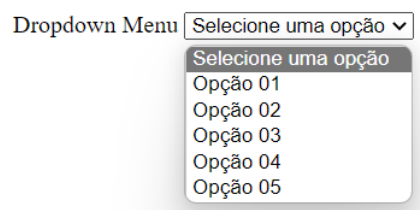


Figura 4.4: Tipos de elementos `<input>` dentro de um formulário html.

4.2.2 A Linguagem CSS

Nós falamos na parte inicial do nosso estudo sobre HTML, mas especificamente na parte do cabeçalho, que uma das referências que normalmente fazemos é a de uma **Cascading Style Sheet (CSS)**. A ideia por trás disso é que a manutenção e o desenvolvimento da aplicação web fica mais simples quando trabalhamos todo o aspecto de estilo visual em um arquivo separado (.css) do arquivo que trata da estrutura da aplicação (.html).

Contudo, na realidade, existem outras formas de trabalhar o visual da aplicação além do arquivo .css em separado. No geral, podemos dizer que existem 3 formas de gerenciamento de estilo de um aplicação web:

- CSS externo - Melhor forma. Nosso material estará focado nesse tipo de arquitetura.
- Bloco interno - As regras ficam no próprio arquivo html. Pode ter aplicações para questões muito específicas. Mas as atualizações vão precisar ser feitas em cada página, sempre que necessário.
- Atributo inline - Pior forma. Aqui, as regras de estilo são definidas diretamente no elemento html. Qualquer mínima alteração terá de ser feita diretamente no elemento e em todas as páginas.

Aqui podemos ver um exemplo de cada aplicação do estilo visual que elencamos acima:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Exemplo CSS</title>

  ###Esse é um exemplo de arquivo externo###
  <link rel="stylesheet" href="style.css" type="text/css">

  ###Exemplo de bloco interno####
  <style type="text/css">
    p {
      font-size: 10pt;
      font-family: "Verdana";
      color: blue;
    }

    h1 { font-size: 16pt;
        font-family: "Impact";
        color: red;
      }
  </style>
</head>
<body>
  ####Exemplo de inline####
  <p style="margin-left: 0.5in; font-size: 8pt;">
    Texto do parágrafo
  </p>
```

```
</body>  
</html>
```

A prioridade de leitura das regras de estilo que o browser vai usar é 1) inline, 2) Bloco interno, 3) CSS externo e 4) Default do navegador.

Sintaxe da linguagem CSS

A leitura de um arquivo CSS é bem simples. A primeira coisa que precisamos saber é quais elementos estão presentes no html que será trabalhado e quais desses elementos possuem atribuição de id específico.

Por exemplo, se tivermos no nosso html dois elementos `<p>`, só que um deles possui o atributo id `<p id="teste">`. Para criarmos uma regra de estilo no nosso CSS basta escrevermos a tag do elemento (sem os símbolos `<>`) do seguinte modo.

```
p {  
    color: red;  
}
```

Essa regra diz que todos os textos contidos nos elementos `<p>` terão a cor vermelha. Contudo, se quisermos adotar uma regra específica para apenas um elemento em questão, podemos definir a regra no css diretamente para o elemento com o seu id.

```
#teste {  
    color: black;  
}
```

Isso nos dará uma página onde todos os textos dos parágrafos serão vermelhos à exceção do parágrafo identificado pelo `id="teste"`.

Podemos resumir a sintaxe do CSS como sendo:

```
seletor {  
    propriedade_1 : valor_da_propriedade_1;  
    propriedade_2 : valor_da_propriedade_2;  
    ...  
    propriedade_n : valor_da_propriedade_n;  
}
```

Ou seja, para aprender bem CSS, vamos precisar aprender as várias maneiras de selecionar os elementos da página html e as propriedades de estilo que o CSS nos permite manipular na construção das nossas aplicações web.

Seletores de Elementos

Eu já adianto, existem muitos tipos de seletores. Nós precisamos decorar todos os tipos? Evidente que não. O importante é saber que o estilo de uma aplicação pode ser desenvolvido de várias maneiras e que, quanto melhor for o método de organização do CSS, mais fácil será o desenvolvimento e a manutenção da aplicação no futuro. A tabela a seguir é uma referência para os vários tipos de seletores em CSS.

Tipo	Link com HTML	Exemplo de Sintaxe
Elemento	Nome da tag html	<code>p {color:blue;}</code>
Identificador	id dos elementos	<code>#ident {color:blue;}</code>
Classe	Classe dos elementos	<code>.classe {color:blue;}</code>
Atributo	Atributos dos elementos	<code>[atrib] {color:blue;}</code> <code>[id="p01"] {color:blue;}</code> <code>[class~="marked" {color:blue;}</code>
Pseudo-Classe	Situações dos elementos	<code>p:first-of-type {color:blue;}</code> <code>p:nth-child(3) {color:blue;}</code> <code>:hover {color:blue;}</code>
Pseudo-Elemento	Partes de elementos	<code>p::first-letter {color:blue;}</code> <code>p::first-time {color:blue;}</code> <code>p::after {color:blue;}</code>
Universal	Todos os elementos	<code>* {color:blue;}</code>

Podemos ver que existem vários modelos de seletores para os elementos html de um página. Alguns deles são dependente de contexto de interação do elemento. Especialmente, as situações de pseudo-classe são muito úteis para criação de aplicações fluidas e avançadas.

Link para lista de todos os pseudo-elementos e pseudo-classes suportados pelo CSS atualmente: [\[LINK\]](#).

Combinação de Seletores

Podemos usar combinações de seletores para definir as regras de estilo das nossas aplicações web. Essas combinações obedecem a determinadas regras que devem ser seguidas para se obter o resultado esperado. Abaixo segue uma tabela de referência.

Regra	Interpretação
<code>A,B {...}</code>	Aplica a mesma regra em A e B
<code>A.B {...}</code>	classes e ids associados à A e B ao mesmo tempo
<code>A B {...}</code>	Elementos em B que também pertençam a A
<code>A > B {...}</code>	Elementos em B filhos de elementos de A
<code>A + B {...}</code>	Elemento em B próximo irmão de elementos de A
<code>A ~ B {...}</code>	Elementos em B próximos irmãos de elementos de A

Prioridade de Seletores

O processamento das declarações CSS obedecem a ordem em 3 regras:

- O processamento é de cima para baixo. A última declaração é a que prevalecerá.
- Regras específicas são prioridade em relação à regras gerais.
- As declarações marcadas como importantes p `{color: red !important;}` são prioritárias.

Valores e Unidades

Atenção aqui. Entender bem quais unidades podem ser usadas e os tipos de unidades ajuda muito o desenvolvimento de interfaces bem planejadas e responsivas.

Tipo	Medida	Significado	Observação
Absoluto	in	Polegadas	
	cm	Centímetros	
	mm	Milímetros	
	pt	Pontos	
	pc	Paicas	
Relativo	em	Tamanho da fonte	1.2em é equivalente a 120% do tamanho original.
	px	Pixels	Um ponto no display onde a página é exibida.
	%	Percentual	120% é a mesma coisa que 1.2 em.

Cores em CSS

Existem infinitas combinações de cores para a paleta que será usada em qualquer aplicação web. Existem diferentes maneiras de definir quais cores serão usadas em CSS:

- RGB hexadecimal - `#RRGGBB`
- RGB abreviado - `#RGB`
- RGB decimal - `rgb(rrr,ggg,bbb)`
- Palavras-Chaves

Podemos usar qualquer uma dessas codificações para definir as cores que vamos usar no estilo das nossas aplicações web.

Display e Box Model

Um dos aspectos mais importantes na construção de uma aplicação web é a disposição dos elementos. Agora que aprendemos como a linguagem CSS nos fornece uma maneira mais simples de controlar as informações de estilo da nossa página HTML, vamos aprender como controlamos os locais onde os elementos são dispostos.

A propriedade `display` é que determina como um elemento e seus filhos são dispostos na página. Alguns valores dessa propriedade se referem a maneira como o elemento é organizado em relação aos elementos irmãos e alguns valores se referem a maneira como seus elementos filhos são dispostos dentro do elemento pai.

Caso não coloquemos nenhuma informação de `display` nos elementos, eles possuem uma categoria default própria que pode ser do tipo `inline` ou `block`.

Os elementos `inline` são colocados automaticamente um ao lado do outro na mesma linha enquanto existir espaço na tela.

- `<a>`
- ``
- ``
- `<button>`
- `<input>`
- etc

Os elementos `block` sempre ocupam uma linha inteira da página.

- `<div>`
- `<h1> ... <h6>`
- `<p>`
- `<form>`
- `<canvas>`
- `<table>`
- etc

Mais ou menos como nessa imagem abaixo

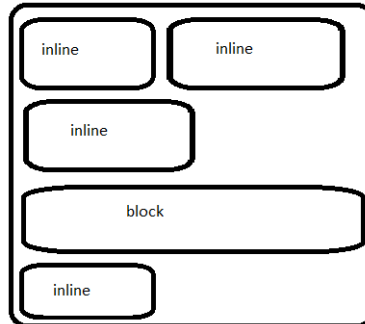


Figura 4.5: Exemplo de elementos inline e block.

Podemos modificar o comportamento padrão de um elemento através do parâmetro `display`: no CSS. Por exemplo, para transformar os `<input>` em um elemento sozinho na página, podemos colocar no CSS a seguinte linha

```
input {  
  display: block;  
  margin: 0 auto;  
}
```

No caso de elementos `inside`⁴, o atributo `display` pode receber os valores `display="table"`, `display="grid"` e `display="flex"`. Quando colocamos esses atributos nos elementos `inside`, o elemento que o contém, que chamamos de elemento pai (`outside`), automaticamente vira um elemento do tipo `display="block"`.

A propriedade `display="table"` em um elemento `outside` permite que os elementos `inside` recebam variações desse atributo para a construção de layout em formato de tabela. Desse modo, se nosso elemento `outside` é do tipo `display="table"`, então, os elementos `inside` podem ser `"table-row"`, `"table-cell"`, `"table-column"`, `"table-caption"`, `"table-row-group"`, `"table-header-group"` e `"table-footer-group"`.

A propriedade `display="flex"` permite que os elementos `inside` sejam controlados de maneira fluida para se ajustar à largura da janela do navegador.

A propriedade `display="grid"` permite um controle das regiões onde os elementos `inside` serão dispostos. Isso dá mais controle ao desenvolver.

⁴também chamados de elementos filhos.

Veremos com mais calma os atributos `display:flex` e `display:grid` porque eles são usados na construção de aplicações mais fluidas e dinâmicas.

Box Model

Existe um conjunto de atributos CSS que compõe o que podemos chamar de **box model**. A ideia aqui é que podemos trabalhar os elementos como pertencentes a uma “caixa” imaginária. Isso torna o design da aplicação mais simples de compreender e também facilita o posicionamento dos elementos ao longo da nossa página.

Os atributos CSS que compõe o modelo de caixa são:

- `margin`
- `border`
- `padding`
- `width`
- `height`
- `background-color`

As propriedades de `margin`, `border` e `padding` aceitam atributos de orientação como `top-right-bottom-left`. Caso queira aplicar o mesmo valor para todos é só informar um único valor no atributo. Se quiser discriminar, é só apontar os valores na ordem descrita no sentido horário ou usar a propriedade inteira para cada lado. A imagem abaixo deixa mais fácil a compreensão dos atributos do modelo de caixa.

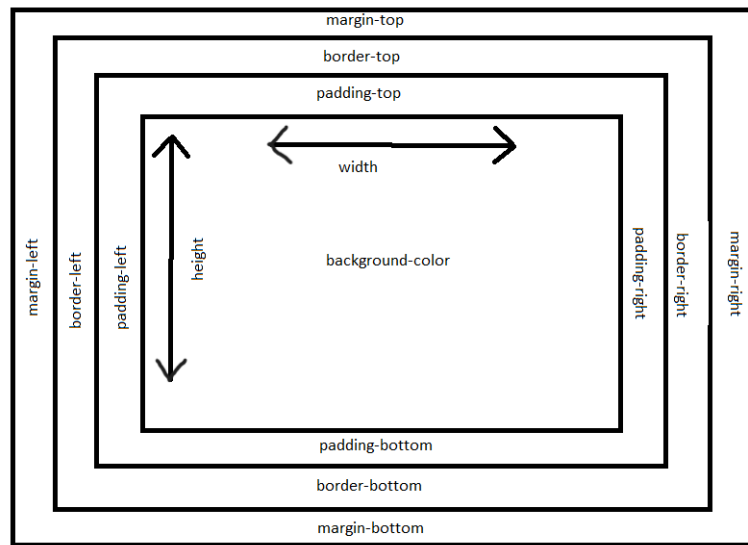


Figura 4.6: O Box Model do CSS.

Durante a elaboração da interface não é nada incomum usar as bordas como método de visualização. O comando que cria a borda é

```
border: solid 20px black.
```

Fundo de Elementos (Background)

Todo elemento html possui um atributo de background que pode ser acessado pelos seguintes comandos:

- **background-color** - Cor de fundo
- **background-image** - Imagem ou gradiente⁵
- **background-repeat** - Comando caso a img não seja do tamanho do elemento. Pode ser do tipo **repeat**, **repeat-x**, **repeat-y**, **space**, **round**
- **background-position** - Define a posição inicial da imagem. Pode ser do tipo **top**, **left**, **right**, **center**, **bottom**

Propriedades de Texto

Existem várias propriedades quando o assunto é texto em CSS. Abaixo podemos ver uma tabela para referência.

⁵Você pode pesquisar para saber a lista dos gradientes disponíveis.

Propriedade	Descrição	Valores
font-family	Tipo de fonte	aria helvetica sans-serif
font-size	Tamanho da Letra	xx-small x-small small medium large x-large xx-large smaller larger 10px 80%
font-style	Estilo da fonte	normal italic oblique
font-weight	Largura	normal bold bolder 100
font-variant	Minúsculas como maiúsculas menores	normal small-caps
line-height	Altura	normal 1.6 80%
color	Cor do texto	[cor]
text-align	Alinhamento	left right center justify
text-shadow	Sombra	x y z [cor] x - horizontal y - vertical z - difusão da sombra
text-decoration-line	Tipo de decoração	underline overline line-through
text-decoration-style	Tipo de linha de decoração	solid wavy dashed dotted double
text-decoration-color	Cor da linha de decoração	[cor]
letter-spacing	Espaçamento das letras	normal 2px 0.1em
word-spacing	Espaçamento das palavras	normal 2px 0.1em
text-transform	Tipo de escrita do texto	capitalize uppercase lowercase

Fontes de Texto na Web

O CSS nos dá as seguintes opções de letras: serif, sans-serif, monospace, cursive e fantasy. Contudo, nós nunca teremos certeza se o navegador do user terá a capacidade de carregar a fonte que desejamos. Para evitar esse problema, podemos definir opções de fontes do seguinte modo:

```
p {
  font-family: "Trebuchet MS", Verdana, sans-serif;
}
```

O navegador do user vai tentar renderizar a página usando a primeira opção, caso ele não consiga, ele vai para as outras opções.

Além das opções padrão CSS, podemos usar fontes proprietárias de outras fontes (Google Fonts, DaFont, Adobe e etc). A maneira de fazer isso é definir uma propriedade de importação como no exemplo abaixo

```
@import url('https://fonts.googleapis.com/css?family=Baloo');

div {
```

```
font-family: 'Baloo', cursive;
}
```

Layouts Responsivos

Não é nada incomum acharmos sites que respondem dinamicamente ao tamanho da tela. Agora vamos aprender um pouco sobre esse método de desenvolvimento de aplicações web.

O Responsive Web Design (RWD) é a ferramenta que define o layout de um site de modo dinâmico ao tamanho da tela ou janela do dispositivo. Para poder usar esse método, nós precisamos planejar nosso código HTML e CSS de maneira compatível com essa metodologia.

Os principais padrões de layout responsivos são. Por enquanto eu vou deixar esse seção mais enxuta:

- Mostly Fluid
- Column Drop
- Layout Shifter
- Off Canvas
- Tiny Tweaks

Media Queries

As media queries são os parâmetros usados na aplicação que usam alguma característica do dispositivo onde a página está sendo exibida. Abaixo nós podemos ver um exemplo de elemento HTML com media query.

```
<head>
  <link rel="stylesheet" media='screen and (min-width: 900px)' href="tela_g.css">
  <link rel="stylesheet" media='screen and (max-width: 600px)' href="tela_p.css">
</head>
```

Nesse exemplo HTML, podemos ver como, de acordo com o tamanho da tela, o arquivo de estilo CSS carregado vai ser o "tela_g.css" ou o "tela_p.css".

Do lado do CSS, a sintaxe das media queries são usadas da seguinte maneira:

```
body { background-color: red; }
```

```
@media screen and (min-width: 600px) {  
  body {background-color: orange;}  
}  
  
@media screen and (min-width: 800px) {  
  body {background-color: yellow;}  
}
```

Podemos ver que, de acordo com a largura da tela, o CSS envia para o navegador uma cor de fundo do body diferente. Agora estamos começando a ver a lógica por trás dos designs responsivos.

As opções de **media types** são:

- all - Qualquer tipo de mídia
- handheld - Para telas responsivas ao toque
- print - Impressoras
- screen - Telas de computadores, smartphones e tablets
- outras

As opções de **media features** são as características dos dispositivos tais como:

- color - Profundidade de cores em bits
- color-index - Número de cores indexadas
- width e height - Largura e altura do viewport
- device-width e device-height - Largura e altura do dispositivo
- orientation - Proporção do viewport (portrait ou landscape)
- resolution - Resolução de saída em dpi

Resolução e Viewport

Quando as tela mudam de tamanho, o valor do **pixel** também é alterado. Para resolver esse problema, o CSS utiliza um método de cálculo que padroniza as medidas independentemente do tamanho da tela.

Se nossa aplicação for desenvolvida para uma tela com 1920 pixels (full HD), podemos converter cada pixel em uma nova unidade que permita a aplicação recalculer os tamanhos dos componentes em pixels de modo a se

adequar melhor ao display. No exemplo a abaixo, nós estamos “mudando” o valor padrão do pixel para caber em uma tela com 1/3 de 1920 (640 pixels):

$$Viewport = \frac{\text{Resolução}}{\text{Pixel-Ratio}} = \frac{1920}{3} = 640 \text{ pixels}$$

Para habilitar esse método de ajuste, o HTML precisa ter a seguinte linha no **head**:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

A Vantagem dessa abordagem é que ela permite a manutenção da leitura quando nossa página é carregada por telas menores. Também podemos controlar a capacidade de rolagem e zoom do usuário por meio dessa meta tag.

Layout Flex

Já aprendemos como reduzir a escala da nossa aplicação com o viewport. Mas, em telas de smartphones ou monitores pequenos, simplesmente reduzir a aplicação para caber no dispositivo pode não ser suficiente para uma boa experiência.

No Layout Flex (flexbox) nós podemos definir o comportamento dos elementos html filhos dentro de um bloco maior. Nesse modelo, nós conseguimos mudar o posicionamento relativo dos elementos filhos sempre que a tela se comportar de determinada maneira prevista (como o caso do nosso site ser aberto em uma tela de smartphone ao invés de um monitor).

Para usar esse recurso, usaremos no elemento pai⁶ o parâmetro **display: flex; flex-wrap: wrap;**. Além de definirmos o tipo de display no elemento pai, usaremos a media query para ajustar o tamanho ideal dos elementos na tela. Podemos ver melhor no exemplo de código abaixo:

```
----- Parte HTML -----
<!DOCTYPE html>
<body>
  <main class='container'>
    <div id="orange"></div>
    <div id="green"></div>
    <div id="yellow"></div>
  </main>
</body>
</html>

----- Parte CSS -----
.container {
  display: flex;
  flex-wrap: wrap;
}
```

⁶Que no exemplo abaixo será um elemento da classe "container"

```
div {  
    height: 80px;  
    width: 100%;  
}  
  
/* tela pequena */  
#orange {  
    background-color: orange;  
    order: 1;  
}  
  
#green {  
    background-color: green;  
    order: 2;  
}  
  
#yellow {  
    background-color: yellow;  
    order: 3;  
}  
  
/* tela media */  
@media screen and (min-width: 600px) {  
    #orange { width : 100% }  
    #green { width : 70% }  
    #yellow { width : 30% }  
}  
  
/* tela grande */  
@media screen and (min-width: 1000px) {  
    #orange { width : 40% }  
    #green { width : 40% }  
    #yellow { width : 20% }  
}
```

O resultado desses códigos acima produzem o seguinte resultado:

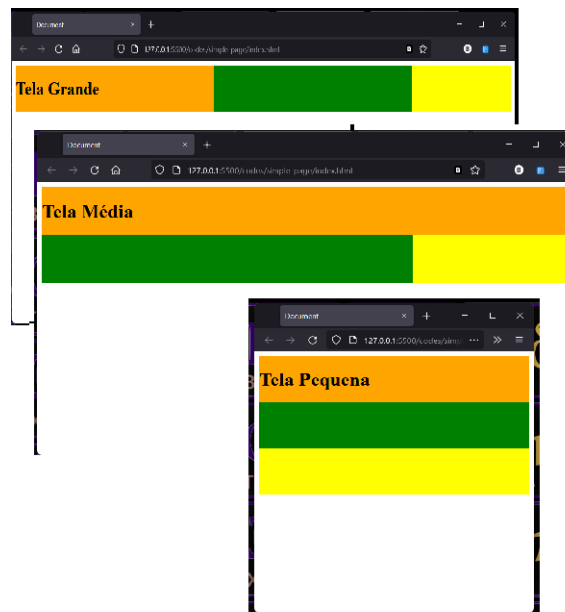


Figura 4.7: Layout Flex com Media Queries

Layout Grid

Para além das medias queries e layout flex, podemos construir o front end de uma aplicação usando o Sistema Grid que o CSS possui. A ideia é pensar no front end da aplicação em termos de dois elementos visuais: O Container e os Itens.

Comentário: Depois eu vou revisitar essa seção com base no material disponível nesse [\[link\]](#).

O sistema Grid possui alguns conceitos que nos ajudam a criar e manter a interface de uma aplicação que use essa metodologia:

- Line - Separa as cells
- Cell - É uma unidade encapsulada em uma linha e uma coluna
- Area - Conjunto de cells
- Track - Um conjunto linear de cells (uma linha ou uma coluna do grid)

Abaixo temos o código de uma aplicação simples usando esse sistema de construção de front end:

```
----- Parte HTML -----
<body>
```

```
<div class="container">
  <header>Header </header>
  <main>Main</main>
  <nav>Sidebar</nav>
  <footer>Footer</footer>
</div>
</body>

----- Parte CSS -----
body {
  background-color: rgb(255, 255, 255);
}

.container {
  height: 700px;
  display: grid;
  grid-template-columns: 20% 30% 30% 19%;
  grid-template-rows: auto;
  grid-template-areas:
    "header header header header"
    "main main main sidebar"
    "footer footer footer footer";
  column-gap: 5px;
  row-gap: 5px;
}

header {
  grid-area: header;
  background-color: orange;
  height: 100px;
}

main {
  grid-area: main;
  background-color: blue;
  height: 500px;
}

nav {
  grid-area: sidebar;
  background-color: red;
  height: 500px;
}

footer {
  grid-area: footer;
  background-color: green;
  margin: solid black 5px;
  height: 100px;
}
```



Figura 4.8: Layout Grid

Com base nessa lógica, podemos posicionar elementos usando o sistemas de coordenadas do CSS Grid. Vamos refazer a interface que acabamos de ver usando apenas esse conceito de “items” dentro de um “container”.

Ao invés de definir a separação da tela como parâmetro da classe container no CSS, nós vamos criar o container e orientar, dentro de cada item, como ele se comportará no grid.

```
----- Parte HTML -----
<body>
  <div class="container">
    <div class="item-a">Header</div>
    <div class="item-b">Main</div>
    <div class="item-c">Nav</div>
    <div class="item-d">Footer</div>
  </div>
</body>

----- Parte CSS -----
body {
  background-color: rgb(255, 255, 255);
}

.container {
  height: 700px;
  display: grid;
  grid-template-columns: 24% 25% 25% 25%;
  grid-template-rows: 10% 80% 10%;
  grid-column-gap: 5px;
  grid-row-gap: 5px;
}

.item-a {
  background-color: orange;
  grid-column-start: 1;
  grid-column-end: span 4;
```

```
    grid-row-start: 1;
    grid-row-end: 1;
}

.item-b {
    background-color: blue;
    grid-column-start: 1;
    grid-column-end: span 3;
    grid-row-start: 2;
    grid-row-end: 2;
}

.item-c {
    background-color: red;
    grid-column-start: 4;
    grid-column-end: 4;
    grid-row-start: 2;
    grid-row-end: 2;
}

.item-d {
    background-color: green;
    grid-column-start: 1;
    grid-column-end: span 4;
    grid-row-start: 3;
    grid-row-end: 3;
}
```

Nem vale a pena mostrar uma imagem do resultado porque ele é exatamente igual à imagem anterior.

Com isso, podemos ver que é o desenvolvimento de uma interface de aplicação web pode ser feito de diferentes maneiras, mas o que realmente importa é uma boa documentação e um planejamento bem feito para que o cliente termine com o que ele realmente precisa e o time de desenvolvimento não precise perder preciosas horas no bem conhecido ciclo de “vai-e-volta” até que o cliente aceite algum layout.

Exemplo de Aplicação Web Responsiva

Para finalizar nosso estudo de desenvolvimento web com CSS vamos fazer uma aplicação simples. O protótipo no MarvelApp pode ser visto nesse [link](#).

O código que gerou esse resultado pode ser visto no repositório desse projeto nesse [link](#).

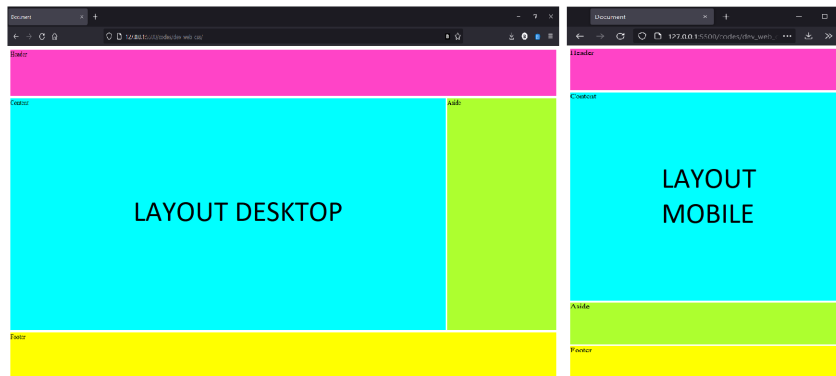


Figura 4.9: Aplicação com front end responsivo

Comentário: Acredite, com os conceitos aprendidos até agora nós já conseguimos fazer aplicações simples com páginas estáticas com relativa facilidade. Uma disso é que enquanto escrevo esse parágrafo⁷ eu acabei de criar a primeira página da aplicação web front end que meu grupo precisa entregar como projeto do primeiro semestre da graduação. Eu usei, basicamente, apenas o que aprendemos aqui e alguma pesquisa no google para coisas mais simples.

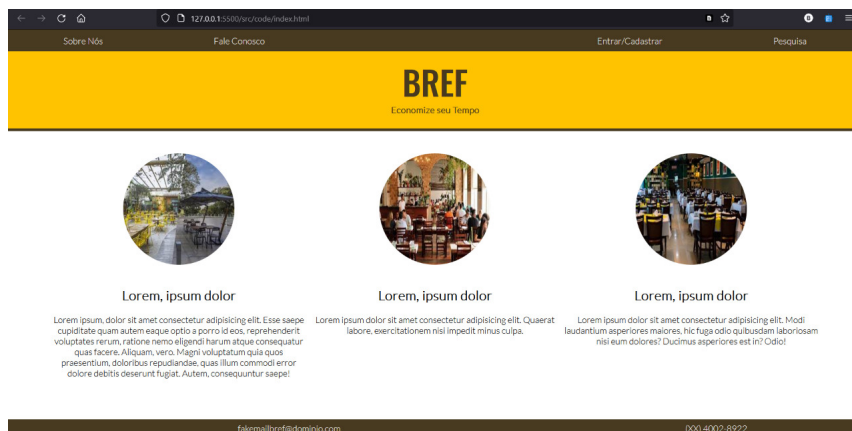


Figura 4.10: Página inicial usando o sistema grid e viewport.

Frameworks front-end - Bootstrap

Como era de se esperar, os programadores perceberam que era possível combinar várias práticas do mercado em “pacotes”, “bibliotecas” ou “frameworks” que, no fundo, são códigos escritos em HTML, CSS e JavaScript. Essas ferramentas facilitam demais o processo de desenvolvimento mas, como tudo na vida, precisam ser usadas com moderação e habilidade.

⁷Dia 29/04/2022.

Não podemos correr o risco de virarmos “escravos” de nenhum framework.

Com o devido aviso dado, podemos iniciar nosso estudo de Frameworks famosos com o Bootstrap. A culpa de várias sites que acessamos terem um “jeitão” parecido é, em boa parte, desse framework que até hoje é o mais famoso. Foi desenvolvido em 2011 por Mark Otto e Jacob Thornton no Twitter⁸ e disponibilizado no GitHub.

O Bootstrap se ajusta automaticamente a diferentes modelos de tela porque tem, dentro dele, tamanhos pré-determinados de telas que se ajustam ao dispositivo⁹. A “fronteira” entre os tamanhos de tela são os chamados **Breakpoints**. A tabela abaixo resume as relações entre tamanho da tela e ajuste do container da aplicação. O site oficial pode ser acessado nesse [link].

	Extra Small <57px	Small ≥ 576px	Medium ≥ 768px	Large ≥ 992px	Extra Large ≥ 1200px
Max container size	(auto)	540px	720px	960px	1140px
Nome da Classe	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-

Download do Bootstrap

Existem duas opções de download desse Framework:

- CSS e JS compilados - Já vem pronto pra uso mas é um pouco limitado.
- Código Fonte - Permite customização e vem com as fontes do Bootstrap mas precisa de compilação.

Para saber como fazer basta uma pesquisa rápida no [google] ou no [youtube].

Conteúdo do Bootstrap

A melhor fonte sobre qualquer tecnologia geralmente é a própria documentação oficial. Mas podemos dividir as partes do Bootstrap em:

- Reboot - Para uniformização da aparência em diferentes navegadores
- Tipografia - Textos e fontes
- Code - Linhas de código
- Images - Relacionado aos recursos de imagens

⁸Isso mesmo.

⁹Igual a nossa aplicação usando viewport.

- Tables - Relacionado aos recursos de tabelas
- Figures - Imagens com textos associados

Componentes do Bootstrap

Nós já sabemos que o HTML possui uma lógica de estruturação da página. Contudo, quando estamos usando o Bootstrap temos que “reaprender” como pensar nossa estrutura da página a partir dos conceitos e modelos do framework.

Para o Bootstrap, a página é criada usando-se **componentes**. A lista é bem grande mas podemos destacar alguns:

- Breadcrumb - Trilha com o caminho do site até uma página
- Navbar - Barra de menu e pesquisa
- Carousel - Conjunto rotativo de imagens em destaque no site
- Cards - Cartões para uso diverso. Podem ser imagens ou texto
- Modal - Caixas de diálogo (tipo um popup só que dentro da página)

Sistema Grid no Bootstrap

Como nós já sabemos, o sistema grid é uma maneira conveniente de controlarmos a estrutura visual da nossa aplicação. Diante disso, não é surpreendente que esse framework também use uma versão dessa lógica.

Para o Bootstrap, **todas as páginas** serão sempre divididas em 12 colunas. Quando um elemento está dentro de outro, o padrão de 12 colunas se mantém. A lógica de divisão é como na imagem abaixo:

.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1
.col-md-8								.col-md-4				
.col-md-4				.col-md-4				.col-md-4				

Figura 4.11: Colunas no Bootstrap

O código HTML usando o Bootstrap é fortemente baseado em `<div>`. A novidade está no uso de classes específicas que dão a lógica contida nos arquivos usados pelo framework.

A hierarquia é similar ao grid system porque a aplicação existira dentro de um container. O Container terá pelo menos uma linha (row). Cada linha terá pelo menos uma coluna. Internalize essa hierarquia de Container, Linhas e Colunas.

```

----- HTML -----
<div class="row">
  <div class="col-md-6" id="cel1">.col-md-6</div>
  <div class="col-md-6" id="cel2">.col-md-6</div>
</div>
<div class="row">
  <div class="col-md-2" id="cel1">.col-md-2</div>
  <div class="col-md-10" id="cel2">.col-md-10</div>
</div>
<div class="row">
  <div class="col-md-9" id="cel1">.col-md-9</div>
  <div class="col-md-3" id="cel2">.col-md-3</div>
</div>

----- CSS -----
#cel1 {
  color: white;
  background-color: red;
  height: 100px;
  border: solid black 3px;
}

#cel2 {
  color: white;
  background-color: blue;
  height: 100px;
  border: solid black 3px;
}

```

Na próxima imagem podemos ver o resultado desse código. O sistema de 12 colunas md-1 torna bem simples a disposição dos conteúdos na tela.



Figura 4.12: Container, Rows e Columns

Exercício Prático - Site de Wired usando Bootstrap

Para exercitar esses conceitos, podemos usar como referência a página do famoso site de notícias Wired¹⁰. A imagem de referência será a versão desktop e a mobile do site.

¹⁰Acesso em 02/05/2022.

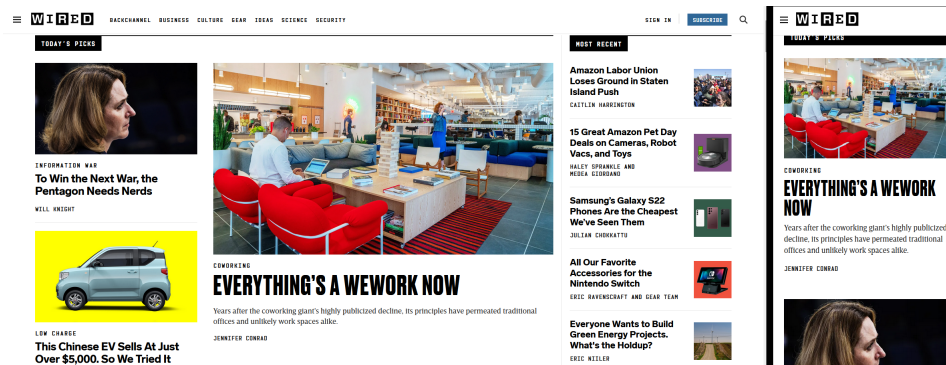


Figura 4.13: Layout Desktop e Mobile

A primeira coisa que precisamos é identificar as estruturas de linhas e colunas da visão desktop. Podemos ver que temos uma barra superior. Uma estrutura de 3 colunas sendo as duas primeiras referentes à uma seção e a lateral direita referente a outra seção.

Em termos dos componentes do bootstrap, podemos fazer uma interface equivalente como na imagem abaixo.

Comentário: Eu to com prazo curto agora por causa do cronograma das aulas então vou ter que deixar esse desafio para depois.

4.2.3 A Linguagem JavaScript

Já aprendemos que a Web utiliza a linguagem HTML para estrutura do conteúdo, a linguagem CSS para formato e apresentação. Contudo, nós sabemos que hoje em dia as aplicações possuem funções bem complexas com sistemas inteiros baseados todo no navegador do client. A linguagem JavaScript é justamente o mecanismo que trás essa capacidade de interatividade e processamento para à web.

Hoje em dia, a linguagem JavaScript (também chamada de JS) é mantida pela ECMA International. Uma organização suíça criada em 1961 cujas atividades são separadas em vários campos de atuação. Um desses campos é a padronização de linguagens de programação. Atualmente, ela elabora os padrões para as linguagens ECMAScript (TC39), C# (TC49) e outras.

Sim, é isso mesmo. Nada de JavaScript. O JS foi criado em 1995 por Brendan Eich mas em 1996 foi transferido para a ECMA para padronização. O primeiro padrão JS dentro do ECMAScript foi feito em 1997.

Como consta na edição 2020 do padrão “O ECMAScript é baseado em várias tecnologias cujas mais conhecidas são o JavaScript (Netscape) e JScript (Microsoft)”. O ECMAScript foi adotado na maioria dos navegadores desde o final dos anos 90. Hoje em dia, praticamente todos os navegadores adotam o padrão ECMAScript e, conseqüentemente, o JavaScript dentro deles.

A ECMA International cria os padrões e especificações que permitam a execução de uma determinada linguagem. O JS é a implementação dessas padronizações dentro dos navegadores do mercado. Tanto é assim, que diferentemente do que acontece com as outras linguagens como Python, R e etc, você não precisa instalar um interpretador ou um compilador para rodar códigos em JS. Os navegadores possuem, dentro deles, um motor próprio que executa o padrão ECMAScript.

Para citar alguns desses motores:

Navegador	Motor Web	Motor ECMAScript
Firefox	Gecko	Spider Monkey
Chrome	Blink	Google V8
Safari	WebKit	JavaScriptCore
IE	Trident	Chakra Core
Edge	EDGE	Chakra Core
Opera	Blink	

Comentário: Não confunda a linguagem de programação Java com JavaScript. São linguagens totalmente diferentes.

Aplicação da Linguagem JavaScript

Da mesma maneira que vimos com o CSS, podemos trazer o JS para nossa aplicação web de diferentes maneiras:

- Por arquivo externo - Código é mantido em um arquivo separado

```
<script type="text/javascript" src="script.js"></script>
```

- Em Bloco interno - Código fica em um bloco dentro do HTML

```
<script type="text/javascript">
    /* Código JS */
    alert("Olá mundo!");
</script>
```

- Inline - Código fica dentro de um atributo do elemento HTML

```
<p onClick="alert('Click feito!');"></p>
```

Agora que sabemos que podemos “turbinar” nossas páginas web com o uso de JS, podemos nos perguntar o que pode ser feito com essa ferramenta. A resposta é praticamente qualquer coisa! Podemos elencar como principais aplicações:

- Manipulação de objetos e tratamento de eventos relacionados aos elementos HTML a partir do uso de uma API chamada DOM (Documento Object Model)¹¹
- Comunicação com servidores e utilização de APIs via AJAX¹² usando o XMLHttpRequest ou na API Fetch
- Armazenamento de dados no client com o uso das APIs Indexed DB e LocalStorage/SessionStorage
- Usar as APIs do HTML5: Canvas, Media, File, Drag and Drop, Geolocation, Web Workers, History

JavaScript além do Browser

As pessoas gostaram tanto de JS que foram capazes de construir uma aplicação que implementa o v8 do chrome fora do browser. Isso mesmo, podemos rodar aplicações em JS direto no terminal do seu computador. Basta usar a aplicação Node.js. Hoje em dia, dá pra fazer uma aplicação inteira, backend e frontend com JS.

Com JS podemos também construir aplicações desktop usando uma biblioteca chamada Electron.

Esse aqui é apenas o começo da nossa caminhada!

Variáveis e Tipos de Dados

A linguagem JS é de tipagem dinâmica. Então o tipo de variável é definido pela própria linguagem na hora da atribuição do valor. Temos dois tokens de atribuição em JS **var** ou **let**.

```
var variavel01;      // Aqui eu declarei sem atribuir nada
var x = 10;          // x agora é um numérico de valor 10
let y = "Olá mundo" // y é uma string
```

¹¹Sério, dá pra gerar praticamente uma página HTML com o uso do DOM. Isso é muito poderoso.

¹²Veremos mais pra frente o que é isso.

Agora temos que entender o motivo de termos dois tokens de atribuição. Mas, para poder explicar isso, precisamos entender que existem diferentes **escopos de variáveis** quando um programa em JS é executado. Os escopos das variáveis em JS são:

- Escopo Global - Variáveis sempre disponíveis para consulta e edição
- Escopo Local - Variáveis existem apenas dentro de um bloco de código (um loop ou uma condicional, por exemplo)

As variáveis declaradas com **var** são de **escopo global** se forem declaradas fora de funções¹³. Já a atribuição feita com **let** é de **escopo local**.

Abaixo temos um exemplo bem bacana mostrando essas diferenças.

```
var a = 5;
var b = 10;

if (a === 5) {
  let a = 4;
  var b = 1;

  console.log(a);
  console.log(b);
}

-> Mostra 4 e 1 no console

console.log(a);
console.log(b);

-> Mostra 5 e 1 no console
```

Comentário: Uma boa prática em JS (e em praticamente todas as linguagens) é sempre declarar as variáveis na parte de cima de cada bloco ou no começo do código.

Também é possível usar o token **const** para atribuição. Esse token funciona igual ao **let** e é definido para constantes. Uma boa prática da comunidade é usar letra maiúsculas para definir suas constantes.

Tipos e Estruturas de Dados

¹³Nesse caso são de escopo local

Todas as variáveis que são números, textos (strings) ou valores booleanos (true ou false) são o que chamamos de **tipo Primitivo**. As variáveis especiais do tipo nulo (null) ou indefinido (undefined) são possuem características de tipo primitivo mas podem ser entendidas como sendo de um tipo único especial. No ECMAScript 6 foi inserida uma nova categoria de variáveis chamadas de **tipo Simbólico ou Symbol**¹⁴. O que não for de tipo primitivo, null, undefined ou symbol, será do **tipo Objeto** que possui uma lista de propriedades, que por sua vez, possuem um nome e valores associados.

Podemos resumir o que acabamos de ver como:

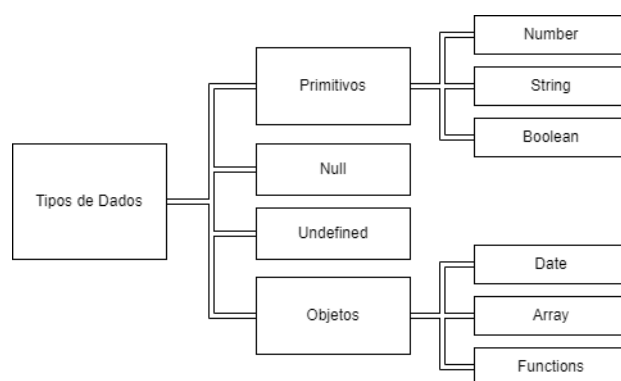


Figura 4.14: Tipos de Dados do JavaScript

Tipo Numérico

JS não faz diferença entre números inteiros ou fracionados. Para essa linguagem, tudo é considerado o que chamamos de **ponto flutuante** de 64 bits. O valor máximo possível é acessado pela propriedade do objeto Number por meio das propriedades `Number.POSITIVE_INFINITY` e `Number.NEGATIVE_INFINITY`.

Quando tentamos fazer uma operação matemática não possível (como raiz de -1) recebemos o resultado **NaN** que significa Not-a-Number.

Não é incomum termos que converter caracteres do tipo texto mas que representam números. Para isso, podemos usar a função nativa `parseInt` ou `parseFloat`.

Tipo Boolean

Os operadores booleanos são operadores que apenas admitem duas entradas:

¹⁴Não vamos nos aprofundar nisso agora. Depois eu atualizo esse material com o ECMAScript 6.

true ou **false**. Podemos chamar essas entradas diretamente ou através de operações lógicas usando operadores lógicos ou matemáticos. Abaixo temos uma tabela que podemos usar de referência.

Operador	Exemplo	Observação
Mesmo valor	<code>x == y</code>	true se mesmo valor
Mesmo valor e tipo	<code>x === y</code>	true se mesmo valor e tipo
Diferente valor	<code>x !== y</code>	true se diferente valor
Maior que	<code>x > y</code>	true se x maior que y
Menor que	<code>x < y</code>	true se x menor que y
Maior igual	<code>x >= y</code>	true se maior igual
Menor igual	<code>x <= y</code>	true se menor igual
Negação	<code>!x</code>	true se x for false
And	<code>x && y</code>	true se x e y forem true
Or	<code>x y</code>	true se x ou y forem true

Tipo String

Qualquer aplicação precisa ser capaz de lidar com textos. Uma string é exatamente uma cadeia de caracteres não numéricos (mas caracteres numéricos também podem compor uma string) que pode ser declarada por meio de aspas simples “abc” ou compostas ‘abc’.

Também é comum precisarmos juntar diferentes strings. No javascript, podemos fazer isso simplesmente usando o operador de soma:

```
str = 'abc' + 'def'

console.log(str)

-> abcdef
```

Alguns caracteres das nossas strings podem ser os mesmos usados na própria linguagem. Isso nos obriga a termos que aprender como fazer um “bypass” desses caracteres. Abaixo temos uma tabela para referência.

Código	Significado
\0	Null
\'	Aspas simples
\"	Aspas compostas
\\	Barra invertida
\n	Nova linha
\r	Retorno
\v	Tabulação Vertical
\t	Tabulação
\b	Backspace
\f	form feed
\uXXXX	Unicode
\xXX	Latin-1

A partir do ECMAScript6 existe outra maneira de compor strings: por meio do uso da crase 'isso é uma **string**'. A vantagem desse método, chamado de **TemplateString** é que esse tipo de string é especial que pode receber expressões interpretáveis como no exemplo abaixo:

```
nome = "bruno"
console.log(nome)
-> bruno

linha = 'meu nome é ${nome}'
console.log(linha)
-> meu nome é bruno
```

Podemos ver que na última linha o resultado faz uso do valor da variável **nome**. Provavelmente, sempre que a gente entra em um site que possui aquela mensagem amigável de “Seja bem vindo, Fulano”, o programador usou essa função para produzir aquele texto.

Tipo Objeto

Objetos são uma boa parte do que faz o JS ser tão poderoso. Cada objeto é simplesmente uma coleção de pares nome-valor. Os nomes são as **propriedades** do objeto e os valores podem ser qualquer tipo de variável que vimos logo antes (incluindo outros objetos e funções). Quando a propriedade for uma função, dizemos que essa função é um **método** do objeto.

```
var objeto1 = new Object(); // Criando o objeto "objeto1"

objeto1.name = "bruno"; // modo 1 de criar uma propriedade
```

```
objeto1["age"] = 28;    // modo 2 de criar uma propriedade

console.log(objeto1)
-> { age:28 , name:bruno }

var objeto2 = {        // modo de criar o objeto
  nome: "bruno",        // junto das propriedades
  idade: 28
}
console.log(objeto2)
-> { nome:bruno, idade: 28 }
```

Para acessar o valor de uma propriedade em um objeto basta escrever algo como: `objeto.propriedade`.¹⁵

Datas e Horas

Na elaboração de uma aplicação é comum ter que lidar com variáveis do relacionadas ao tempo. Para isso o JS possui um tipo de objeto especialmente projetado para facilitar essa tratativa. O objeto `Date()`. Podemos usa-lo como:

```
var a = new Date() // Atribui a data atual ao objeto
var c = new Date(string) // Atribui a data por uma string
var d = new Date(Ano,Mes,Dia,Hora,Segundo,Milisegundos)
```

Geralmente estamos preocupados apenas com o dia, mês e ano. Para criar um objeto com esses valores basta ir colocando na mesma ordem do último elemento do exemplo acima.

Aviso: Quando você for indicar o mês, saiba que para o JS, janeiro é 0 e dezembro é 11. Então quando quisermos salvar a data '01/01/2022' usaremos `Date(2022,0,1)`.

Como todo objeto, a variável criada com o `Date()` possui várias propriedades e métodos. Uma boa referência é o material do MDN que pode ser visto [aqui]. Mas podemos elencar alguns métodos importantes como:

```
var z = new Date()

z.getFullYear()
z.getMonth()
```

¹⁵Python é assim também.

```
z.getDate()
z.getDay()
z.getHours()
z.getMinutes()
z.getSeconds()
z.getMilliseconds()
z.getTimes()
```

Acho que não é necessário explicar cada método desse porque o nome é bem alto explicativo.

Arrays, Vetores ou Matrizes

Um array é uma estrutura de dados semelhante a uma lista. Em JS temos algumas maneiras de criar arrays:

```
/* Modo 01 */
var a = new Array()
a[0] = 'valor01'
a[1] = 'valor02'

/* Modo 02 */
var b = new Array('valor01','valor02')

/* Modo 03 */
var c = ['valor01','valor02']
```

Todos os 3 modos produzem o mesmo resultado e podem ser usados alternadamente.

Como qualquer objeto, existem vários métodos muito úteis dentro de um `Array()`.

```
concat() // Junta dois ou mais vetores
fill() // Preencher os elementos em um vetor com um valor estático
find() // Retorna o valor do primeiro elemento em um vetor que atender ao filtro
findIndex() // Retorna o índice do primeiro elemento em um vetor
forEach() // Chama uma função para cada elemento do vetor
indexOf() // Busca um elemento no vetor e retorna a sua posição
isArray() // Verifica se um objeto é um vetor
join() // Junta todos os elementos de um vetor em uma string
lastIndexOf() // Pesquisar o vetor por um elemento, começando no final
pop() // Remove o último elemento de um vetor e retorna o elemento
push() // Adiciona novos elementos para o final de um vetor
reverse() // Inverte a ordem dos elementos em um vetor
slice() // Seleciona uma parte de um vetor e retorna o novo vetor
sort() // Classifica os elementos de um vetor
```

```
splice() // Adiciona/remove elementos de um vetor  
toString() // Converte um vetor em uma string e retorna o resultado  
valueOf() // Retorna o valor primitivo de um vetor
```

Comentário: Eu achei estranho que o material não falou nada sobre vetor e matriz. Então uma hora eu volto para expandir essa parte com esses outros dois tipos de dados.

Controle de Fluxo

Em qualquer linguagem de programação¹⁶ existem maneiras de organizar blocos de código para o compilador/interpretador saber o que priorizar na hora do processamento do código. Em JS isso é feito com o uso das chaves `{ }`.

Declarações de Seleção

Quando estamos fazendo um script de código, é muito comum termos que nos adaptar as situações onde determinados eventos podem ou não acontecer. Para isso, usamos algumas estruturas lógicas que permitem o processamento de blocos de texto apenas se alguma condição predeterminada seja satisfeita. Ou seja, usamos um condicionante do tipo “Se-Então”.

```
if (expressao) {  
    bloco de codigo caso true  
}  
else {  
    outro bloco de codigo caso false  
}
```

Existem situações onde existem múltiplas possibilidades. Pensando nisso, o JS possui um operador de `switch` que permite vários condicionantes de maneiras mais simples do que vários ifs sucessivos.

```
switch (expressao) {  
    case valor01:  
        bloco_se_01  
        break  
  
    case valor02:  
        bloco_se_02  
        break
```

¹⁶Sempre vai existir alguma exceção, eu sei.

```
    default:
      bloco_se_nao_01_ou_02
      break
  }
```

Veja que usamos o token **break** para indicar que vamos para outra condicional. Não podemos esquecer disso.

Abaixo temos um exemplo de uma aplicação simples usando esse método de gestão de fluxo. Não se preocupe em entender o código inteiro, foque apenas no que estamos estudando.

```
// Função que recebe um input no terminal
const readLine = require('readline').createInterface({
  input: process.stdin,
  output: process.stdout
})

readLine.question('Qual seu nome? ', name => {
  switch (name) {
    case 'bruno':
      console.log('OI, BRUNO!')
      break

    case 'mario':
      console.log('Salve, Mario!')
      break

    default:
      console.log('Olá, ${name}. Seja bem-vindo(a)!')
  }
  readLine.close()
})

> Qual seu nome? bruno
> OI, BRUNO!

> Qual seu nome? mario
> Salve, Mario!

> Qual seu nome? pedro
> Olá, pedro. Seja bem-vindo(a)!
```

Declarações de Repetição

Além de termos tokens que nos permitem executar apenas alguns blocos de código. Existem outros que nos permitem usarmos um mesmo bloco de código repetidamente. Chamamos essas estruturas lógicas de laços de repetição.

A primeira maneira de criarmos um laço de repetição em JS é por meio do token `while` do seguinte modo:

```
// Exemplo de loop com while
var i = 0

while (i <= 5) {
  console.log('Contagem: ${i}')
  i++ // soma + 1 ao valor de i
}

> Contagem: 0
> Contagem: 1
> Contagem: 2
> Contagem: 3
> Contagem: 4
> Contagem: 5
```

Primeiro criamos a variável `i` e usamos a estrutura de loop `while` para a seguinte regra “Se `i` for menor igual a 5 então mostre a mensagem ‘Contagem: Número `i`’”. Quando o valor da variável `i` é 6, a expressão do loop retorna um `false` e saímos do loop.

O operador que aprendemos anteriormente faz um loop enquanto a condição expressa entre os parênteses não for `false`. Mas existem situações onde queremos executar o bloco de código um número definido de vezes. Para isso o JS possui o token `for`.

```
// Exemplo de loop com for
let frutas = ['pera', 'uva', 'maca', 'banana']

for (let index = 0; index < frutas.length; index++) {
  console.log(frutas[index])
}

> pera
> uva
```

```
> maca  
> banana
```

Nesse exemplo acima vemos que o parênteses da expressão do **for** possui 3 linhas de códigos separadas por ponto e vírgula. Na primeira, criamos uma variável `index`. Na segunda, temos a nossa expressão que avalia se o `index` é menor que o tamanho do array `frutas`. E na terceira, temos a regra de incremento `+ 1` para cada iteração do loop.

Dentro do loop, nós mandamos ele lançar no terminal o valor do elemento do array correspondente ao `index`.

Funções

Nós aprendemos anteriormente que funções são umas das variáveis do tipo objeto no JS. Sempre que pudermos generalizar um tratamento de dados por meio da criação de uma função, devemos optar por esse caminho porque assim tornamos a manutenção dos nossos programas melhor e evitamos ter que repetir linhas de código muito parecidas.

Mesmo sendo um objeto, o JS possui um token específico para a criação de funções. Abaixo temos dois exemplos adaptados do material do curso.

```
// Criacao de uma funcao de soma  
function soma(x,y) {  
    var total = x + y  
    return total  
}  
  
soma(2,3)  
  
> 5
```

Para o segundo exemplo, nós queremos construir uma função que retorna a média dos valores apresentados entre os parênteses. Para isso, precisaremos de um dos tokens que vimos na seção passada. Sabemos muito bem que a média é calculada pela soma dos n valores dividida pela quantidade n . Para isso, teremos que usar o token de laço de repetição.

```
function media() {  
    var soma = 0  
    n = arguments.length
```

```
        for (var i = 0; i < n; i++ ) {  
            soma = arguments[i] + soma  
        }  
        return soma / n  
    }  
  
media(2,3,4,5)  
  
> 3.5
```

Calma, eu sei que em uma primeira vista é estranho esse objeto **arguments** ter sido chamado do nada. Acontece que no JS (e em muitas linguagens) alguns objetos, a exemplo das funções, possuem métodos que podem ser usados mesmo no momento da sua criação. Nesse caso, é o exemplo desse objeto que nos diz a quantidade de argumentos que foram inseridos na nossa função `media()`.¹⁷

Também podemos usar as funções para criação de (estruturas padronizadas de objetos) que são conhecidas como *classes*. Abaixo vamos criar uma estrutura para objetos do tipo **Pessoa**.

```
function Pessoa (primeiro,ultimo) {  
    this.primeiro = primeiro  
    this.ultimo = ultimo  
  
    this.nomeCompleto = function() {  
        return this.primeiro + ' ' + this.ultimo  
    }  
  
    this.nomeCompletoInvertido = function() {  
        return this.ultimo + ' ' + this.primeiro  
    }  
}  
  
var chefe = new Pessoa ('Bruno','Ruas')  
  
console.log(chefe.nomeCompleto)  
console.log(chefe.nomeCompletoInvertido)  
  
> Bruno Ruas  
> Ruas Bruno
```

¹⁷Legal né?!.

Novamente, temos um termo sendo usado sem que antes ele tenha sido criado. Nesse caso é o termo **this**. Ele é usado para fazer referência ao objeto criado (ou classe) como podemos ver no chamamento dos valores do `console.log`.

Outra novidade é que criamos uma “cópia” do objeto **Pessoa** só que com um outro nome: **chefe**.¹⁸

Arrow Functions

Agora vamos entrar num ponto que, para mim, foi bem difícil de entender no começo. O JS permite que a construção de **funções anônimas**, ou seja, funções que não precisam de nomes definidos. Mas antes de aprendermos mais sobre isso, precisamos saber que existem diferentes modos de se criar uma função em JS. Abaixo temos 3 maneiras de se criar a função **soma**.

```
// forma tradicional
soma = function(a,b) { return a + b }

//forma com arrow function
soma = (a,b) => {return a + b}

//arrow function com chaves omitidas
soma = (a,b) => a + b
```

Agora que entendemos um pouco melhor a construção de funções em JS, podemos ir mais fundo nas **arrow functions**. Esse método de definição de funções foi criado para facilitar a criação de funções dentro de contextos, ou seja, se você está criando uma função em uma linha de código normal (sem estrair em um bloco de código) é melhor usar a forma tradicional. Agora, se você estiver em um contexto diferente (como em um parâmetro de um objeto ou mesmo uma função) é melhor usar a arrow function.

Assim como os outros objetos, existem propriedades e métodos das arrow functions que podemos usar. Um exemplo disso é o operador **this** que faz referência ao bloco em que nossa arrow functions está contida. Em uma função normal, nós aprendemos que esse operador faz referência à própria função. Abaixo temos um exemplo adaptado do material.

```
// usando 'this' em uma arrow function
var Pessoa2 = {
  nome: 'Bruno',
  amigos: ['Ana', 'Clarck', 'Bruce'],
```

¹⁸Sério, tudo isso é bem legal. Mas vai demandar um tempo até se acostumar.

```
    exibeAmigos() {  
        this.amigos.forEach(f => console.log(this.nome + ' é amigo de ' + f))  
    }  
}  
  
Pessoa2.exibeAmigos()  
  
> Bruno é amigo de Ana  
> Bruno é amigo de Clarck  
> Bruno é amigo de Bruce
```

Vamos ver o que esse código acima nos ensina. Primeiro nós criamos uma variável de objeto chamado **Pessoa2** e atribuímos duas propriedades a ela: **nome** e **amigos**. Sendo que a propriedade ‘**amigos**’ é um array com 3 elementos. Como **Pessoa2** é um objeto, nós podemos criar métodos dentro dele (que nada mais são do que funções em um objeto), e é exatamente o que fizemos com a função **exibeAmigos()**.

Quando criamos o método **exibeAmigos**, nós usamos o operador **this** em um contexto onde ele faz referência ao bloco que contém a função, nesse caso, é a variável **Pessoa2**. Na primeira linha da nossa função **exibeAmigos** nós usamos um método

Dentro de função **exibeAmigos** nós chamamos a propriedade **amigos** do nosso objeto **Pessoa2** por meio do ‘**this**’. Como já vimos, um array é um objeto em JS, portanto, ele possui várias propriedades e métodos dentro dele. O **forEach** que está ali é precisamente uma desses métodos desse objeto que faz um loop para cada elemento do array.

Dentro do loop criado pelo método **forEach** do nosso array **amigos** nós criamos uma arrow function que usa a variável **f** (que nada mais é do que o elemento do array no loop). Então nós fizemos o seguinte, para cada elemento do array que agora chamamos de **f**, vamos fazer um **console.log** que nos dá o nome da **Pessoa2**, a string “ conhece ” e o nome do amigo que é a variável **f**.

Com isso temos o resultado apresentado no final desse bloco de código. Com o tempo as coisas vão ficar menos confusas, mas já podemos ver que o JS possui bastante metodologias de processamento de informação que teremos que dominar para tirar o máximo das nossas aplicações web.

Documento Object Model (DOM)

Como o contexto de aplicação do JS sempre foi o web, essa linguagem possui algumas integrações ao ambiente do navegador que são muito úteis para a construção de aplicações verdadeiramente inteligentes. Uma das capaci-

dades mais relevantes é a de alterar estruturas HTML e CSS por meio de códigos escritos em JS. Isso é feito por intermédio da API Document Object Model (DOM) que é um padrão da W3C para os navegadores. Abaixo temos os principais elementos que a compõe.

Todos esses objetos estão vinculados ao objeto maior `window` que é a janela do navegador.

- `history`
- `navigator`
- `location`
- `screen`
- `document`
 - `link`
 - `anchor`
 - `form`
 - * `button`
 - * `check box`
 - * `radio`
 - * `password`
 - * `reset`
 - * `submit`

Cada um desses elementos possui informações e podem ser manipulados para conseguirmos construir as aplicações da maneira como quisermos.

Comentário: No material do curso nós só aprofundamos em dois desses vários elementos. Depois eu volto aqui a medida que for aprendendo mais sobre os outros.

Objeto Window

O objeto `window` é o representante da janela do browser. Ele contém toda a hierarquia que mostramos nessa lista de elementos acima. Além de conter todos esses elementos dentro dele, esse objeto¹⁹ possui métodos e propriedades que são úteis para o gerenciamento das nossas aplicações.

¹⁹Como o próprio nome diz, tudo no DOM são, em algum grau, objetos. Mantenha isso em mente!

Na parte de **armazenamento de dados** o window nos dá duas maneiras: **localStorage** e **sessionStorage**. No primeiro os dados são mantidos mesmo se o navegador seja fechado. Já o sessionStorage mantém os arquivos apenas enquanto o navegador é mantido aberto. Abaixo temos um exemplo de como usar esses repositórios.

```
// guardando dados no repositório de sessão
sessionStorage.setItem('login','Bruno Ruas')
alert('O usuário logado é: ' + sessionStorage.getItem('login'))

> Aparece um popup com a mensagem "O Usuário logado é: Bruno Ruas"
```

Veja que para salvar o dado, temos o uso do par 'nome'-'valor' e do método **setItem**. Para obter o dado, basta usar o método **getItem** e passar o nome do valor salvo anteriormente.

Também podemos atrelar algumas funcionalidades ao tempo. É bem comum vermos depois de x minutos a seção de um site ser encerrada, por exemplo. O objeto window possui algumas maneiras de lidarmos com o tempo:

- **setInterval(funcao, intervalo)** - Browser executa uma função continuamente a cada x milissegundos
- **clearInterval()** - Cancela a repetição da função
- **setTimeout(funcao,intervalo)** - Agenda a execução de uma função com um delay de x milissegundos
- **clearTimeout()** - Cancela o agendamento

Object Document

Logo após o navegador processar os arquivos da página web, o objeto **document** passa a existir e pode ser manipulado pelo nosso código JS. No material temos uma lista das principais propriedades que esse objeto possui²⁰.

²⁰A descrição eu só vou colocar se for algo não óbvio.

Propriedade	Descrição
<code>addEventListener</code>	Uma função dispara se um evento ocorre
<code>baseURI</code>	Retorna a URI
<code>body</code>	Retorna ou modifica o body
<code>cookie</code>	Retorna todos os cookies
<code>characterSet</code>	Charset da página
<code>documentElement</code>	Todo o html
<code>documentURI</code>	URI do document
<code>forms</code>	Um array com os forms do html
<code>getElementsById</code>	
<code>getElementsByClassName</code>	
<code>getElementsByName</code>	
<code>getElementsByTagName</code>	
<code>images</code>	
<code>lastModified</code>	Data de modificação do documento
<code>links</code>	Array com todos os links
<code>querySelector</code>	Primeiro elemento por um seletor CSS
<code>querySelectorAll</code>	Array com todos os de um seletor CSS
<code>removeEventListener</code>	
<code>scripts</code>	Array com os scripts do documento
<code>title</code>	
<code>URL</code>	

Não tem muito pra onde correr. Só vamos aprender bem usando, mas por agora, basta termos em mente que essas funcionalidades existem e poderão ser úteis em algum momento.

Abaixo temos um exemplo usando uma função JS para alterar um elemento do body através do DOM.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>PUC-MG</title>
  <!-- JavaScript-->
  <script type='text/javascript'>
    function Executar() {
      document.getElementById('saida').innerHTML = Date()
    }
  </script>
</head>
<body>
  <button type="button" onclick="Executar()">
```

```
        Aperte!  
    </button>  
  
    <h1>Saída</h1>  
    <div id="saida">Condição Inicial</div>  
</body>  
</html>
```

No código acima temos o script em JS dentro do próprio arquivo HTML usando a tag `<script>`. Nesse script nós criamos a função `Executar()` que nada mais faz do que procurar o elemento HTML cujo id é igual a 'saida' e substitui o HTML desse elemento pela data atual vinda da função nativa `Date()`. Abaixo podemos ver a diferença entre a situação inicial e a final após apertar o botão.

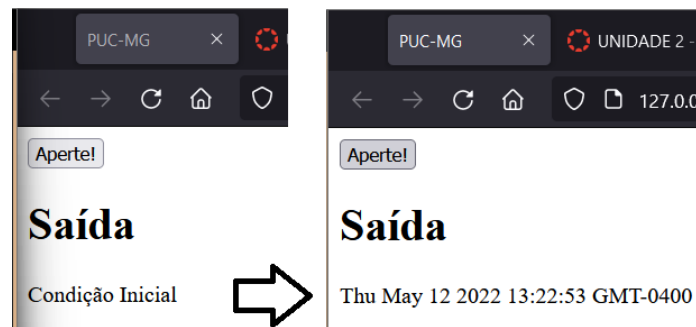


Figura 4.15: Manipulação do HTML com DOM e JS

Objeto Navigator

Como o próprio deixa claro, o objeto navigator representa o software do navegador usado pelo user. Esse objeto nos dá algumas informações úteis sobre o navegador usado. Abaixo temos uma tabela com alguns dos métodos contidos nesse objeto.

Propriedade	Descrição
<code>appName</code>	Código do navegador
<code>appVersion</code>	Nome do navegador
<code>cookieEnabled</code>	Versão do navegador
<code>geolocation</code>	Cookies habilitados
<code>language</code>	Geolocation
<code>onLine</code>	
<code>platform</code>	Se o browser está online
<code>product</code>	Qual sistema operacional
<code>userAgent</code>	Engine do navegador
	User-agent que o browser envia ao navegador

Abaixo temos um exemplo de página web adaptado do material do curso que faz uso o objeto navegador e retorna a geolocalização do navegador.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>PUC-MG</title>
  <script type="text/javascript">
    // funcao que recebe as coord e retorna um string
    function showPosition(position) {
      lat = position.coords.latitude
      long = position.coords.longitude
      texto = 'Lat: ${lat} e Long ${long}'

      document.getElementById('saida').innerHTML = texto
    }

    // funcao que muda o HTML do elemento saida
    function getPosition() {
      if (window.navigator.geolocation) {
        window.navigator.geolocation.getCurrentPosition(showPosition)
      } else {
        x.innerHTML = 'Não tivemos acesso a' +
          'sua localização!'
      }
    }
  </script>
</head>
<body>
  <button type="button" onclick="getPosition()">
    Geolocation
  </button>
  <h1>Mensagem</h1>
  <div id="saida">Sua localização vai aparecer aqui!</div>
</body>
</html>
```

Nesse código acima temos uma página simples com um botão que, ao ser ativado, dispara a função `getPosition()`. Essa função dispara uma outra

função chamada `showPosition`. O resultado pode ser visto na imagem abaixo²¹.

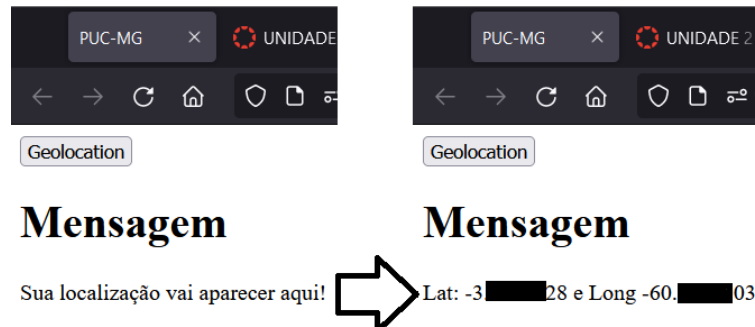


Figura 4.16: Utilização do Objeto Navegador

Eventos

Outra funcionalidade interessante do DOM é a capacidade de usarmos não apenas os elementos da página mas também o contexto (ou eventos) em que esses elementos se encontram. Por exemplo, é comum vermos uma imagem aumentar de tamanho quando colocamos o mouse sobre ela. Com essa funcionalidade, conseguimos explorar vários cenários de interação do usuário com a página.

No exemplo abaixo, temos o uso de dois contextos do mouse sobre um elemento de texto. Quando o mouse está sobre o elemento, nós mudamos a cor dele para vermelho. Quando o mouse não está sobre o elemento, a cor dele se torna preto.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>PUC-MG</title>
</head>
<body>
  <p onmouseover="this.style.color='red'"
    onmouseout="this.style.color='black'">

    Texto

  </p>
</body>
</html>
```

²¹Eu só dei uma censurada porque segurança nunca é demais.

Podemos ver mais um exemplo do uso do token `this` mas dessa vez fora de um bloco de código de uma função. Nesse contexto, esse operador faz referência ao elemento HTML e altera o seu valor do parâmetro de estilo que indicamos (a cor do texto).

Mesmo sendo possível fazer uso do atributo de evento direto no HTML, como fizemos no exemplo anterior, a boa prática é manter essas features em um arquivo JS à parte (igual nós fizemos com o CSS) pois isso torna o arquivo de estrutura mais fácil de ler. Abaixo temos exatamente a mesma funcionalidade mas usando o campo de script do meta mas que poderia ser hospedado em um arquivo `.js` em separado.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>PUC-MG</title>
</head>
<body>
  <p id="texto">
    Texto
  </p>
</body>

<script type="text/javascript">
  var elem = window.document.getElementById('texto')

  elem.addEventListener('mouseout',
  function () {
    window.document.getElementById('texto').style.color = ''
    window.document.getElementById('texto').style.fontSize = ''
  }, false)

  elem.addEventListener('mouseover',
  function () {
    window.document.getElementById('texto').style.color = 'red'
    window.document.getElementById('texto').style.fontSize = '20px'
  }, false)
</script>
</html>
```

Veja que primeiro nós criamos uma variável `elem` utilizando o `getElementById`. Como essa variável é um objeto, podemos usar um de seus métodos chamado `addEventListener` que recebe 3 parâmetros: o primeiro é o gatilho de disparo, o segundo é a função que será executada e o terceiro é, para todos os efeitos, sempre falso.²²

No exemplo acima temos dois gatilhos de eventos cadastrados. O primeiro é o `mouseout` que nada mais é que o padrão. O segundo é `mouseover` que dispara sempre que o mouse estiver sobre o elemento selecionado.

²²Você pode pesquisar um pouco se estiver curioso do motivo disso.

Mesmo que a primeira vista pareça mais simples usar os atributos de contexto direto no HTML, nós precisamos pensar em situações onde a aplicação se torna demasiada grande. Nesses casos, é muito melhor, para controlar a evolução da nossa aplicação web e garantir uma boa manutenção, mantermos em arquivos separados tanto o estilo quanto a funcionalidade.

Outra coisa importante a ser notada é que o script deve estar abaixo da variável que será manipulada. Primeiro o elemento precisa existir para usarmos o nosso código. Esse é o motivo da tag `script` estar na parte de baixo da página e não no `head` como nos outros exemplos.

Na tabela a baixo temos os principais tipos de gatilhos que podem ser usados pelo DOM. Para usar algum desses no direto no HTML é necessário colocar a palavra ‘on’ na frente. Por exemplo, ‘click’ vira ‘onclick’.

Propriedade	Descrição
<code>click</code>	Click em link ou elemento
<code>change</code>	Default alterado em input texto
<code>focus</code>	Foco em um elemento
<code>blur</code>	Ao tirar o foco
<code>mouseover</code>	Mouse em cima
<code>mouseout</code>	Mouse em outro lugar
<code>select</code>	Select em um form
<code>submit</code>	Submit em um form
<code>resize</code>	Mudança na janela do browser
<code>load</code>	Algum elemento é carregado
<code>unload</code>	Ao sair da página

Comentário: Aqui o professor da disciplina faz o desafio de criarmos uma calculadora simples com o uso dos elementos estudados nessa seção. Um dia eu volto aqui e faço esse desafio.

A Notação de Objetos (JSON)

O JavaScript Object Notation (JSON) é um formato de descrição de dados que se baseia em texto e pode ser lido diretamente sem muita dificuldade. Existem vários outros tipos de escrita de dados onde, geralmente, temos um trade-off entre eficiência versus simplicidade. Quanto mais próximo da linguagem de máquina, melhor é pro computador processar mas mais difícil é para os humanos entender.

Como o JSON se popularizou muito, diversas outras linguagens já possuem a capacidade de processar dados nesse formato. O que aumenta ainda mais a aceitabilidade dele como veículo de envio e codificação de informação tanto

no frontend quanto no backend.

Falando em backend, com a popularização de programas como nodejs, é cada vez mais comuns a construção de comunicação entre a camada de aplicação web e servidor através de APIs cuja atividade é, simplificada, receber JSON e enviar JSON.

Como dito anteriormente, o JSON não é o único formato existente no mercado. Para destacar alguns outros, temos o XML, RDF, Planilhas, CSV, Documentos TXT, JPEG-2000, TIFF e vários outros formatos proprietários. Mas, sem dúvida, a competição mais árdua é entre JSON e XML.

Sintaxe do JSON

Um arquivo JSON é muito parecido com um bloco de código JS. Começa e termina com chaves {} e cada elemento é formado por um par de nome-valor e são separados por vírgula. O JSON permite diferentes tipos de valores que podem ser dos mesmo tipos de dados que estudamos anteriormente para o JS. Abaixo temos um exemplo de um arquivo JS.

```
{
  "idade": 28,
  "nome": "Bruno Ruas",
  "materias": [
    {
      "materia": "Econometria",
      "professor": "Bill Gates"
    },
    {
      "materia": "Microeconomia",
      "professor": "Steve Jobs"
    }
  ],
  "nerd": true
}
```

Nesse exemplo, podemos ver que o valor para Idade é do tipo number. Nome é uma string. Matérias é um array, veja que ele está entre colchetes [], exatamente como aprendemos antes. Cada elemento do nosso array de matérias é um objeto, que começam e terminam com chaves, com duas propriedades: matéria e professor. Por fim, temos um booleano para a pergunta "É nerd?".

Os outros tipos de dados como datas, geolocalização e outros, são passados em JSON como uma string. Tempos que lembrar disso se um dia

precisamos lidar com esse tipo de informação.

Comentário: Quando estamos criando um JSON dentro de um código JS nós não precisamos colocar as aspas no nome dos elementos. Mas é padrão que essas aspas sejam usadas em objetos JSON. Então temos que ficar atentos a isso também.

Objeto JSON no JavaScript

Agora que sabemos um pouco sobre esse tipo de estrutura de dados, vamos aprender como manipular esse objeto dentro de um script JS. A maneira que o JS tem que trabalhar com JSON é por meio de um objeto nativo da linguagem chamado, nem um pouco sem querer, de JSON. Esse objeto possui dois métodos úteis: Um `parse()` que recebe uma string e retorna um objeto na notação JSON e um `stringfy()` que pega um objeto análogo JSON e transforma em string.

```
// JSON em string
var TextoJSON = '{"Revistas": [
  {
    "titulo":"V de Vingança",
    "autor":"Frank Miller"
  },
  {
    "titulo":"Batman - The Dark Knight",
    "autor":"Frank Miller"
  },
  {
    "titulo":"One Piece Nº 29",
    "autor":"Eichiro Oda"
  }
]}'

// Usando o metodo de parse do JSON
var banca = JSON.parse(TextoJSON)

function listaTitulos() {
  lista = ''
  for (let i = 0; i < banca.Revistas.length; i++) {
    lista += banca.Revistas[i].titulo + ' - ' +
      banca.Revistas[i].autor + '\n'
  }
  console.log(lista)
}
```

```
listaTitulos()

> V de Vingança - Frank Miller
> Batman - The Dark Knight - Frank Miller
> One Piece Nº 29 - Eichiro Oda
```

Primeiro nós criamos uma string no formato de um JSON. Depois passamos essa string pelo objeto JSON com o método `parse`. Após isso, nós temos um objeto chamado `banca` com a mesma estrutura do nosso JSON desejado. Depois nós criamos uma função de loop que retorna uma string com o título, o autor e o caracter especial de quebra de linha `\n`.

Para converter nosso objeto novamente para string basta usarmos o seguinte comando:

```
JSON.stringify(banca,null,2)
```

O primeiro parâmetro é o objeto a ser convertido. O segundo nós podemos colocar como `null`²³. Por fim, o terceiro diz o tamanho do espaço para melhorar a indentação.

Saber como lidar com objetos JSON dentro de um script JS nos permite construir aplicações inteiras apenas com JS. Um framework muito famoso hoje em dia chamado MEAN (MongoDb, Express, Angular e NodeJS) utiliza o JS como principal linguagem para construção de toda a aplicação web.

Programação Ajax

Diferente do que vimos até agora, o **Asynchronous Javascript and XML** ou AJAX, não é uma tecnologia propriamente dita mas sim uma técnica de programação que utiliza diferentes tecnologias. A ideia geral é construir ferramentas que possuam as seguintes características:

- Páginas com padrão XHTML e CSS
- Dinâmica através do DOM
- Troca de informações por JSON, XML ou outro
- Recuperação assíncrona de dados com o objeto XMLHttpRequest ou APT fetch
- JavaScript como linguagem

²³Existe uma explicação para isso mas não precisamos dela agora.

Algumas aplicações muito famosas foram construída usando, em algum grau, essa abordagem. Podemos citar o Youtube, Gmail, Google Earth e mais um monte de outras aplicações do Google.

Para entendermos como o AJAX é diferente da abordagem tradicional, vamos comparar as duas maneiras. No modelo tradicional, o browser faz requisições ao web server que, por sua vez, devolve a página solicitada com as devidas alterações previamente programadas. Contudo, nessa abordagem, toda a inteligência de negócio é mantida no ambiente do servidor. O esquema abaixo nos permite relembrar como é feita a comunicação entre essas duas entidades.

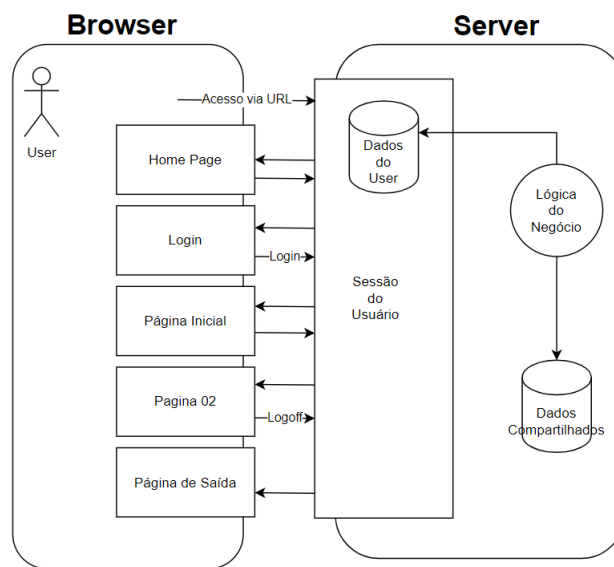


Figura 4.17: Esquema de uma aplicação web tradicional

O problema dessa abordagem tradicional é que os dados enviados entre servidor e browser são páginas inteiras. O método AJAX trouxe a possibilidade de quebrarmos esses dados em "pacotes" menores de informações. Nesse caso, ao invés de requisições de páginas inteiras, podemos pedir ao servidor apenas "pedaços" menores de informação.

A metodologia AJAX trás para o lado do cliente (client side) parte da tarefa de processar os dados. Parte do processamento fica no server side e a outra parte é feita no client. Durante a sessão, várias requisições são feitas do browser (que contém o código JS) para o server por meio do uso do XMLHttpRequest ou API Fetch. A atualização da página acontece via DOM e não pelo recebimento de um HTML novo.

Essa comunicação acontece geralmente por meio de arquivos XML ou JSON (mas pode ser qualquer outro formato de dados).

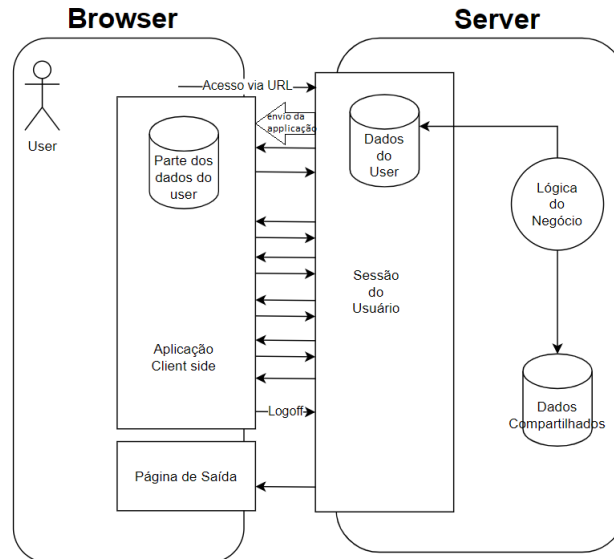


Figura 4.18: Esquema de uma aplicação web com AJAX

Ao abrirmos a aplicação pela primeira vez, o servidor nos envia a maior parte dos dados que serão usados na visualização (a base de toda a aplicação). Depois disso, todas as mudanças na tela serão baseadas em DOM e arquivos de dados JSON ou XML.

Podemos elencar algumas vantagens desse modelo de construção de solução web:

- Redução do tráfego na rede
- Redução de carga no web server
- Flexibilidade no desenvolvimento do lado do server (foco em APIs)

Mas como tudo na vida, o AJAX possui algumas desvantagens:

- Maior complexidade de desenvolvimento
- Aplicação mais pesada no client
- Só funciona em navegadores mais modernos
- Pode precisar de ajustes dependendo do navegador do client
- O usuário perde a opção de avançar e voltar no seu histórico²⁴

²⁴Isso dá pra remediar mas não é simples.

Objeto XMLHttpRequest

Agora que entendemos melhor o que é o AJAX e como podemos construir aplicações com ele, precisamos entender bem uma das suas principais ferramentas: o objeto XMLHttpRequest. Inicialmente criado pelo Microsoft e adaptado pelo Mozilla, o XMLHttpRequest é um objeto (mas também pode ser interpretado como uma API) que é fornecido pelo próprio navegador e que permite que nosso código JS troque dados com um servidor web.

O objeto XMLHttpRequest é a base do AJAX. Mas cuidado. Apesar do nome, ele aceita a troca de informações em diferentes formatos além do XML: JSON, HTML, TXT, XML. Além disso por ele podemos nos conectar à serviços por outros métodos além do HTTP.

Para aprender como usar essa ferramenta, vamos por partes. Podemos começar o nosso script com a criação de um objeto que será do tipo XMLHttpRequest.

```
// funcao caso a conexao funcione
function success() {
    window.document.getElementById('texto').innerHTML = this.responseText
}

// funcao caso a conexao de erro
function error(err) {console.log('Erro:',err)}

// criacao do objeto XmlHttpRequest
var xhr = new XMLHttpRequest()

// executa a funcao success se a requisicao funcionar
xhr.onload = success

// executa um funcao erro se a conexao nao funcionar
xhr.onerror = error

// definindo a requisicao
xhr.open('GET','https://api.github.com/users/brunoruas2')

xhr.send()
```

Esse script acima executa uma consulta a uma url do github que retorna um JSON com informações a respeito da conta de algum usuário. Essa requisição é feita no método `send()`. Em caso de sucesso, o xhr dispara a função `success` que, por sua vez, altera usa o DOM para printar na tela o JSON coletado. Em caso de erro, ele mostra, no console²⁵ com uma mensagem de texto que contém o erro.

²⁵Para acessar basta apertar f12 em qualquer navegador moderno.

Como podemos perceber (e como quase tudo em JS) existem vários métodos úteis no objeto XMLHttpRequest que devemos aprender. Abaixo temos uma tabela de referência com esses métodos.

Propriedade	Descrição
status	Código HTTP da resposta
statusText	Texto da resposta
readyState	Status do pedido
responseText	Txt bruto da resposta
responseXML	Resposta em um objeto no DOM. Mas só funciona se o tipo for text/xml
onreadystatechange	Dispara uma função quando o readyState muda
onerror	Executa função se erro
onprogress	Dispara uma função em caso de demora na resposta
onload	Função se o send funcionar

Para cada requisição, o método readyState retorna um estágio diferente. Podemos pensar que cada requisição possui uma série de passos a serem feitos até a sua conclusão. Esses passos são lidos pelo método onreadystatechange e possuem características que podemos ver na lista abaixo.

Os estágios de uma requisição são:

- **Uninitialized** (0) - Objeto criado mas não iniciado
- **Loading** (1) - Objeto criado mas não usou o método send()
- **Loaded** (2) - Send() executado mas os cabeçalhos não estão disponíveis
- **Interactive** (3) - Alguns dados recebidos mas não completamente
- **Completed** (4) - Todos os dados foram recebidos e podem ser lidos

Abaixo temos um exemplo de uso do readyState para verificação do status da chamada.

```
function requisicaoAJAX() {  
    var xmlhttp = new XMLHttpRequest()  
}  
  
xmlhttp.onreadystatechange = function() {  
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
```

```
        divElement = document.getElementById('texto')
        divElement.innerHTML = xmlhttp.responseText
    }
}

xmlhttp.open("GET","www.google.com/api.php")
xmlhttp.send()
```

Podemos ver que nós criamos uma função disparada apenas quando o status muda e, como colocamos uma condição dentro, só teremos a realm modificação caso o status da comunicação seja 200 (que é o padrão para sucesso) e o status da comunicação seja o de completo.

API Fetch

Os navegadores atuais nos permitem uma alternativa ao XMLHttpRequest chamada API Fetch. A vantagem é que essa nova tecnologia nos permite fazer uso das *promises* que simplifica a escrita da programação assíncrona. Podemos ver um exemplo abaixo.

```
<script>
    fetch('https:api.github.com/users/brunoruas2')
      .then(res => res.json())
      .then(data => console.log(data))
      .catch(err => console.log('Erro: ', err))
</script>
```

Comentário: O material não se aprofunda no tema, então eu volto aqui para aprofundar no futuro. Aqui tem um [link](#) com um material de referência da MDN.

Capítulo 5

Fundamentos de Engenharia de Software

Bibliografia

Bibliografia Básica

- PRESSMAN, Roger S.; MAXIM, Bruce R. Engenharia de software: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016. E-book ISBN 9788580555349. Capítulos 1, 2, 3
- PRIKLADNICKI, Rafael, WILLI, Renato, e MILANI, Fabiano. Meé-todos ágeis para desenvolvimento de software. Porto Alegre: Bookman, 2014 1 recurso online ISBN 9788582602089 Capítulos 1,2,3,8,12,13
- SOMMERVILLE, Ian. Engenharia de software, 10ª ed. Pearson 768 ISBN 9788543024974 Capítulos 1,2,3,4

Bibliografia Complementar

- COHN, Mike; SILVA, Aldir José Coelho Corrêa da. Desenvolvimento de software com Scrum: aplicando métodos ágeis com sucesso. Porto Alegre: Bookman, 2011. E-book ISBN 9788577808199
- LARMAN, Craig. Utilizando UML e padrões: uma introdução á análise e ao projeto orientados a objetos e desenvolvimento iterativo. 3. ed. Porto Alegre: Bookman, 2007. E-book (695 páginas) ISBN 9788577800476
- PAULA FILHO, Wilson de Pádua. Engenharia de software, v. 2 projetos e processos. 4. Rio de Janeiro LTC 2019 1 recurso online ISBN 9788521636748

- VETORAZZO, Adriana de Souza. Engenharia de software. Porto Alegre SAGAH 2018 1 recurso online ISBN 9788595026780
- WAZLAWICK, Raul Sidnei. Engenharia de software conceitos e práticas. Rio de Janeiro GEN LTC 2013 1 recurso online ISBN 9788595156173

5.1 Conceitos e Processos de Software

A engenharia de software é subárea da Ciência da Computação que lida com as atividades de desenvolvimento, operação e evolução de software. Esse campo surgiu com a crise do software de 1968.

5.1.1 Definições

Agora vamos aprender os conceitos usados ao longo do trabalho de engenharia de software:

- Programa - Conjunto de instruções em uma linguagem de programação.
- Software - Programa + Estrutura de Dados + Documentação.
- Sistema - Conjunto de elementos interdependentes de Softwares, Hardware e Pessoas. Podem ser intensivos em qualquer umas dessas 3 partes.

5.1.2 Modelos e Princípios de Processo de Software

O processo de Software é um conjunto de etapas usadas para a produção de soluções de software. Podemos elencar dois conceitos importantes que compõe o processo de software:

- Descrição de Processos:
 - Atividades - Lista de etapas necessárias.
 - Produtos ou Artefatos - Produto gerado pelas atividades.
 - Papéis - Quem executa cada atividade.
 - Condições - As requisições pré e pós execução das atividades.
- Modelos de Ciclo de Vida:
 - Modelo Sequencial Linear
Análise/Projeto/Codificação/Teste.
 - Modelo em Cascata
Definição/Projeto/Implementação/Integração/Manutenção.

- Modelo Incremental
O projeto é quebrado em incrementos e cada incremento possui um modelo sequencial linear ou em cascata.
- Modelo Incremental Evolutivo
Esboço/loopEspecificação/Desenvolvimento/Validação até que se tenha a versão final.
- Modelo Espiral
loopPlanejamento/Modelagem/Construção/Entrega/Feedback para cada incremento novo ao software.
- Modelo Iterativo
É o modelo Sequencial Linear mas com possibilidade de retorno para as etapas anteriores até que se esteja aprovado pelo cliente.
- Modelo V
Durante todas as etapas de processo de software nós já vamos definindo os testes que serão usados para a aprovação do produto.

Hoje em dia, temos dois modelos mais usados. A modelo incremental foca em entregar um pedaço de cada vez e o modelo iterativo permite entregar versões mais simples do produto e ir aprimorando elas. O modelo atual mais usado é justamente o **Modelo incremental iterativo**.

Agora que aprendemos o conceito de modelo de processo de software, vamos analisar algumas abordagens de elaboração de software. Podemos dizer que existem 3 grupos principais de processos de gestão de software: 1) Processos ágeis; 2) Processos Prescritivos e 3) Processos Enxutos (lean process).

Comentário: No material do curso só foram abordadas os dois primeiros processos.

5.1.3 Processos Ágeis

Os processos ágeis nasceram no final do século XX. Seguem o modelo incremental e iterativo de desenvolvimento. Os incrementos são pequenos e sucessivos (2 a 3 semanas). O cliente está constantemente em contato com o produto gerado no ciclo. A documentação é reduzida porque há muita comunicação interpessoal.

Existem várias metodologias mas podemos elencar alguns:

- eXtreme Programming (XP)
- Scrum
- Dynamic System Development (DSDM)

- Feature Driven Development (FDD)
- Crystal Families

Hoje em dia o método mais usado é o Scrum. A novidade dele é que a abordagem do desenvolvimento é empírica e permite a evolução dos requisitos do processo ao longo do processo.

O Scrum é dividido em apenas 3 etapas: 1) Planejamento inicial do projeto; 2) Loop de desenvolvimento e feedback (chamado de sprint) e 3) Entrega ao cliente.

As equipes do scrum são pequenas, multidisciplinares, de liderança diluída e trabalham com um foco de melhorias pequenas em um prazo mais curto (2 ou 4 semanas). Existe a figura do facilitador do processo chamado Scrum Master.

Os requisitos do software são mantidos no artefato chamado **Backlog** e serve de norte para os times de desenvolvimento.

Existem 3 papéis no processo de gestão do Scrum:

- Product Owner (PO) - O cliente ou alguém representante da vontade dele. Podemos pensar no PO como a ponte entre a empresa-cliente e a empresa-desenvolvedora.
- Scrum Master - É o facilitador do time de desenvolvimento. Atua como ponte entre o time de desenvolvimento e o PO. Atenemos para o fato do PO não participar do processo de desenvolvimento técnico.
- Equipe de Desenvolvimento - É auto-organizada e responsável pela produção dos algoritmos que comporão o software.

Agora veremos de maneira organizada os artefatos produzidos no processo de Scrum:

- Backlog do Produto - Lista de características necessárias ao software atreladas a um grau de importância. Cada característica é fruto de uma **história de usuário** que é composta de 3 informações (quem?; o que? e por quê?).
- Backlog da Sprint - É um subconjunto das características elencadas do backlog do produto. Esses itens serão o foco da sprint (2 a 4 semanas).
- Incremento do Produto - É o resultado do trabalho realizado na sprint.

Além dos papéis e dos artefatos, existem as cerimônias do modelo Scrum:

- Reunião de planejamento da Sprint - Decide quais características do Backlog do projeto serão objeto de trabalho pelo time de desenvolvimento.
- Daily - Acompanhamento a cada 24 horas do esforço do time de desenvolvimento para o alcance do planejamento da sprint. Algo rápido (15 min).
- Revisão da Sprint - Avaliação pelo PO do cumprimento do backlog da sprint. Foco no produto.
- Retrospectiva da Sprint - Melhoria do processo por meio de feedback de todas as partes envolvidas no processo de sprint. Foco no processo.

5.1.4 Processos Prescritivos

Antes do predomínio das metodologias ágeis, os processos de controle de produção de software eram orientados por processos prescritivos, também são chamados de processos dirigidos por planos. A ideia é primeiro planejar tudo e ir visualizando o caminhar dos trabalhos em termos do planejamento inicial. O Rational Unified Process (RUP) é o mais famoso desses modelos.

O RUP hoje pertence à IBM e possui algumas características principais:

- Possui vários princípios dos quais podemos citar:
 - Foco nos riscos principais
 - Garantia do valor
 - Permitir mudanças
 - Definição da arquitetura da solução o mais breve possível
 - Construção da solução em componentes
- Baseado em componentes/etapas planejadas
 - Disciplinas (o que deve ser feito)
Requisitos/Análise/Projeto/Implementação/Teste
 - Fases (as etapas de cumprimento das disciplinas)
Concepção/Elaboração/Construção/Transição
- Possui linguagem padronizada: Unified Modeling Language (UML)
- É dirigido por caso de uso
- Funciona por modelo iterativo-incremental

Os benefícios dos processos prescritivos ainda são vistos nas maiores empresas, principalmente relacionados ao uso da UML para definição de etapas necessárias em interações e processos. Abaixo temos um exemplo retirado do material do curso. Mais informações sobre a UML podem ser encontradas nesse [LINK].

```

Especificação do caso de uso Matricular em disciplinas do sistema de controle acadêmico. Fonte: A
especificação do caso de uso foi adaptada do livro BEZERRA, Eduardo. Princípios de análise e projeto
de sistemas com UML. Rio de Janeiro: Campus, 2003.

Matricular em disciplinas
Sumário: O aluno usa o sistema para se matricular em disciplinas.
Ator primário: aluno
Ator secundário: Sistema Financeiro
Pré-condições: o aluno está identificado pelo sistema

Fluxo Principal:

1- O aluno solicita a matrícula em disciplinas;
2- O sistema apresenta a lista de disciplinas disponíveis para o semestre corrente para as quais o
alunos possui pré-requisitos;
3- O aluno seleciona as disciplinas desejadas e solicita a matrícula;
4- O sistema aloca o aluno em turmas de ofertas das disciplinas desejadas e informa ao aluno a turma
alocada para cada disciplina bem como o professor, os horários e dias da semana e as salas de aula;
5- O aluno confirma as alocações feitas;
6- O sistema realiza a matrícula e envia os dados para o Sistema Financeiro;
7- O caso de uso termina.

Fluxo Alternativo (4): Inclusão em lista de espera

a) Se não há vaga ou oferta disponível para alguma disciplina selecionada pelo aluno, o sistema informa
o fato ao aluno e fornece a opção de inserir o aluno em uma lista de espera para aquela disciplina;
b) Se o aluno aceitar o sistema insere o aluno na lista de espera desejada e apresenta a posição do
aluno na lista. O caso de uso retorna ao passo 4;
c) Se o aluno não aceitar o caso de uso prossegue a partir do passo 4.

Fluxo de Exceção (4): Violação de regra de negócio relativa quantidade máxima de créditos

a) Se o aluno já atingiu a quantidade máxima de créditos em que pode se matricular por semestre, o
sistema informa a quantidade de disciplinas que ele pode se matricular e o caso de uso retorna ao
passo 2;

Pós-Condições: O aluno foi inscrito em turmas das disciplinas selecionadas ou foi acrescentado a
listas de esperas das disciplinas selecionadas.

```

Ao final de cada fase são superados os marcos principais do RUP. Cada marco significa o maior risco relacionado àquela etapa. Na fase de concepção é o marco de objetivo de ciclo de vida. Na fase de elaboração é o marco que arquitetura do software. Na fase da construção é o marco da capacidade operacional inicial e, por fim, no marco da transição é o marco da entrega do produto.

Existem problemas nos métodos prescritivos, os principais são:

- Forte apego à hierarquia
- Segmentação elevada do processo de construção
- Em situações críticas, acabam dando lugar a processos ágeis

5.1.5 Quando usar cada Processo?

Na vida real, podemos encontrar vários modelos misturados no dia a dia das empresas. As práticas em cada empresa são orgânicas e fortemente baseadas

na cultura organização local.

Podemos sempre analisar os modelos como uma matriz de 2 eixos: Cascata x Iterativo e Disciplinado x Flexível. Aqui nós só analisamos os processos iterativos. Cabe a você saber se precisa de um processos mais formal como o RUP ou algo mais rápido e flexível como o SCRUM.

5.1.6 Requisitos

Podem ser divididos em 2 grupos: requisitos de cliente e requisitos do software. A primeira classe é focada nas necessidades dos usuários¹ que utilização o sistema (é o problema a ser resolvido). A segunda categoria são as características que o produto deve ter para cumprir os requisitos dos clientes (são as ferramentas que o sistema terá para interagir com os users).

Os requisitos de software podem ser divididos em funcionais e não funcionais. Essa divisão será abordada de maneira mais detalhada abaixo.

5.1.7 Requisitos Funcionais

Os requisitos funcionais são as características que o software deve ter para resolver os problemas elencados como objetivos do sistema proposto. São definidos pelos stakeholders (user, clientes, especialistas e investidores). No SCRUM eles estão no backlog do projeto e no RUP está num documento específico para isso.

É uma lista de exposições breves das funcionalidades que o software fará e como ele se comportará em relação a alguma interação dos usuários. Atente para o fato que os requisitos funcionais são sempre relacionados a algum usuário e não à características técnicas do sistema.

5.1.8 Requisitos Não Funcionais

São as descrições das normas e padrões do produto de software. É aqui que definimos a linguagem de programação, o ambiente, os critérios de segurança, banco de dados, disponibilidade do produto, desempenho e etc.

Um requisito não funcional deve sempre citar um critério de aceitação quantificável. Desse modo, podemos realizar testes objetivos na hora de avaliar se o desenvolvimento da feature foi bem sucedido na iteração.

Podemos elencar alguns tipos de requisito não funcional:

- Desempenho

¹Por meio das histórias de usuários ou dos casos de uso.

- Disponibilidade
- Portabilidade
- Usabilidade
- Capacidade e Degradação
- Manutenibilidade

Outros requisitos não funcionais são relacionados ao processo de desenvolvimento. Como por exemplo:

- Restrição da equipe desenvolvedora
- Qual processo de software deve ser usada
- Qual documentação deve ser criada

Além dessas duas classificações, podemos ter restrições relacionadas ao projeto de software:

- Qual SGBD deve ser usado
- Plataforma de disponibilidade (web ou não)
- Qual linguagem de programação usada
- Qual o SO das plataformas
- Existência de sistema legado

Todos os requisitos não funcionais estarão no backlog da sprint através da aceitação do incremento pelo cliente e no RUP existe uma documentação específica para isso.

5.2 Atividades e Artefatos da Engenharia de Software

O processo de produção de software é dividido em atividades com seus respectivos responsáveis e os artefatos criados a cada etapa finalizada.

As atividades são divididas em técnicas, gerenciais, testes e de apoio². Essas atividades são as que compõe toda a gestão da engenharia de software.

²Não focaremos nessa parte mas são as atividades de RH, administrativo e etc.

5.2.1 Atividades Técnicas

Dentro das atividades técnicas nós temos a **engenharia de requisitos, design/projeto de software, implementação/codificação, testes e aceitação do cliente**.

Podemos elencar as seguintes atividades técnicas necessárias ao bom processo de engenharia de requisitos:

- Levantamento de Requisitos (Elicitação):
 - Entrevistas
 - Observação
 - Leitura de documentação
- Análise dos Requisitos:
 - Análise das lacunas
 - Modelagem gráfica
 - Revisão das descrições
- Especificação dos Requisitos:
 - Descrição sem ambiguidades
 - Linguagem natural, controlada ou específica
- Validação dos Requisitos:
 - Revisão de tudo
 - Prototipagem
 - Notações complexas podem dificultar entendimento do cliente
 - Validação por parte do cliente

Agora vamos ver as atividades de Projeto (design) de Software:

- Ponderação das alternativas de soluções
- Escolha da solução que será implementada
- Detalhamento da solução escolhida (elaboração do projeto):
 - Arquitetura do Software: Alto nível de abstração. Foco nos requisitos não funcionais. Representação das partes gerais da solução.
 - Projeto Detalhado: Baixa abstração. Definição dos objetos e das interações. Foco nos requisitos funcionais. Algoritmos e estruturas de dados.

Uma vez que temos os requisitos elencados e o projeto definido, entramos na etapa de implementação ou codificação.

Implementados os algoritmos, temos a etapa de testes de software para validar os requisitos e garantir que os objetivos sejam alcançados. Podem ser manuais ou automatizados.

Por fim, temos a aprovação do cliente no sentido de cumprimento das funcionalidades esperadas e da qualidade exigida da solução.

Após a aprovação, existe a etapa de manutenção de software que é composta da repetição de todas as etapas expostas acima. Cada manutenção ou aprimoramento passa pelas etapas descritas desde a análise de requisito até a aprovação.

Medidas de Software

São abordagens de medição e definição de metas para o cumprimento das etapas programadas para alcance dos objetivos da solução contratada.

5.2.2 Atividades Gerenciais

São as atividades que atuam no controle da complexidade da solução desenvolvida e podem ser divididas em **gestão de configuração**, **gestão de projeto**, **gestão de requisitos** e **gestão de processos** e, além dessas, possuem atividades afins como gestão da qualidade e estimativas de software.

A gestão de configuração ou gestão de versões é a atividade que cuida da manutenção e organização dos arquivos produzidos durante todo o processo de software. É a atividade que controla as atualizações dos programas e mantém a memória de todas as etapas anteriores.

A gerência de projeto de software é a atividade que controle a dinâmica de tempo, pessoas, custos envolvido no processo de desenvolvimento.

A gerência de requisitos é a atividade de controle das necessidades de mudança no escopo do projeto bem como controla as mudanças na mudança da necessidade do cliente a respeito da mudança de requisitos. Também atua na priorização dos requisitos para a definição dos focos de trabalho. Outra atribuição relacionada é o controle da rastreabilidade dos requisitos pois todas as etapas de elaboração devem ser relacionadas a algum requisito que pertença ao escopo solicitado pelo cliente.

A gestão de Processos é a atividade de definição e melhoria do processo de gestão de software de acordo com as boas práticas, dos modelos de capacitação e maturidade como (CMMI e MPS.BR).

A gestão da qualidade é a atividade que avalia as várias interfaces de dinâmicas que impactam no resultado final do produto de software.

A estimativa de software é a atividade de gerar previsões com base na história da empresa de desenvolvimento afim de melhorar a alocação dos recursos para cumprimento das etapas previstas no início do processo de planejamento.

5.2.3 Testes de Software

O objetivo dos testes é identificar os problemas da solução desenvolvida mas, como tudo na vida, existem restrições a quantidade e qualidade de testes possíveis de serem feitos uma vez que existem custos associados a essa atividade.

Diante das restrições impostas pela realidade e da complexidade do processo de desenvolvimento, é impossível, não importa o dimensionamento do esforço, garantir uma aplicação livre de erros. O foco dessa atividade é garantir que, ao dado nível de confiança requerido, que o software entregará as capacidades requeridas no projeto.

Os testes são necessários para garantir o cumprimento dos requisitos funcionais e não funcionais e podem ser divididos em dois tipos:

- Testes Funcionais/Caixa Preta - Baseados no ponto de vista do usuário do software.
- Testes Estruturais/Caixa Branca - Ponto de vista de quem desenvolveu o software por meio de inputs e avaliação de outputs.

Uma boa maneira de realizar os testes funcionais é reproduzir as situações listadas nas histórias dos usuários.

Uma **Plano de Testes** é o documento que indica o conjunto de informações relacionadas ao teste realizado, tais como:

- Testes de desempenho
- Testes funcionais da história de usuário x
- Teste de responsividade
- Teste de campos de formulários

- Teste de navegabilidade ou links
- Teste de ponta a ponta

No teste ponta a ponta passamos por todas as principais características e funcionalidades do produto que desenvolvimento para cumprimento dos requisitos.

Um plano de teste deve conter os casos de testes que, por sua vez, devem conter as seguintes informações:

- Objetivo
- Valores de entrada
- Valores de saída esperada
- Valores de saída real
- Registro de execução (falha ou sucesso)

5.2.4 Artefatos e Templates

Artefatos

Os **artefatos** são um dos produtos que as atividades técnicas e gerencias produzem em cada ciclo de trabalho e podem ser usados nas etapas posteriores da execução do projeto. Existem vários tipos, vamos elencar alguns:

- Artefatos do processo de desenvolvimento:
 - Backlog do produto³
 - Diagramas de casos de usados
 - Descrição de casos de uso
 - Documento de especificação de requisitos
Descreve os requisitos baseados em casos de uso ou outra forma de descrição.
- Artefatos do processo de gerenciamento:
 - Documento de arquitetura de software
O nome já denuncia mas é importante porque contém vários diagramas do desenho da aplicação e como a solução foi construída em partes funcionais.
 - Plano de Teste de software
 - Casos de testes

³Nós já sabemos o que é.

- Lista de bugs
- Plano de projeto
- Matriz de rastreabilidade
Mostra como os requisitos (que compõe as linhas da matriz) se relacionam com os artefatos produzidos durante o processo de produção.

Templates

Nós já aprendemos o que são os artefatos de software e em que contexto eles são gerados, agora, vamos aprender algumas ferramentas e templates que nos auxiliam no processo de criação desses artefatos durante o processo de desenvolvimento de software.

Comentário: Essa seção é mais para consulta quando você precisar gerenciar algum projeto de desenvolvimento de software. Vou tentar manter os links atualizados mas caso algum deixe de funcionar, pode me avisar pelo twitter.

Backlog do produto e Kanban

Existem várias maneiras de organizar o cumprimento dos requisitos contidos no backlog do projeto. O kanban é um quadro onde transformamos cada item do backlog em unidades separáveis (geralmente post-its ou quadros) onde podemos mover para quadrantes de um board maior. Usualmente temos os quadrantes "A fazer", "Fazendo" e "Feito". Desse modo, podemos ver rapidamente o estado do desenvolvimento das atividades programadas para a sprint.

Existem várias ferramentas virtuais que podem ser usadas no processo como:

- Trello
- PivotalTracker

Especificação de Requisitos de Software

Existem vários templates disponíveis na internet:

- IEE/ISO/IEC 29148
- Esse exemplo
- Método Volere
- PUC-MG

Documento de Arquitetura de Software

- Architecture View Template
- Interface Template
- IEE/ISO/IEC 42020

Nesse link aqui você pode ver como elaborar um caso de teste funcional a partir dos casos de uso.

Aqui tem um template de ata de reunião.

Por fim, temos um template de matriz de impacto de mudanças aqui. E uma matriz de rastreabilidade em excel aqui.

5.2.5 Desenhando Processos de Software

Essa última seção é um exercício onde vamos colocar em prática todos os conceitos aprendidos até agora. Temos que saber que os conceitos aprendidos não são regras imutáveis na aplicação prática em um processo de software. Podemos combinar características de vários modelos durante o processo de execução de um planejamento sempre com foco na melhora contínua da qualidade do software.

Mesmo tendo muita flexibilidade sobre o processo de software, podemos elencar características que são obrigatórias em qualquer desenho:

- Qual o modelo de ciclo de vida
- Quais as atividades que comporão o processo de software e quais as técnicas usadas ao longo delas
- Quais produtos ou artefatos serão gerados a cada etapa
- Os papéis dos agentes relacionados ao longo do processo

Nós começamos o nosso estudo de engenharia de software pelos modelos de ciclo de vida exatamente porque eles regem grande parte das atividades e artefatos produzidos durante todo o processo de software. A maturidade da empresa, dimensionamento da mão de obra, recursos disponíveis, verba do projeto, tempo de execução e outras características são importantes para definição do melhor modelo de ciclo de vida a ser adotado.

A nossa jornada pela engenharia de software vai ser em grande medida construir um amplo repertório de modelos de ciclo de vida, atividades e artefatos.

Para facilitar o complexo processo de software, existem várias ferramentas que centralizam as diferentes etapas e simplificam o processo de gestão:

- Bizagi Modeler
- Eclipse Process Framework

Capítulo 6

Lógica Computacional

6.1 Bibliografia

Bibliografia Básica

- HUNTER, David J. Fundamentos de Matemática Discreta. Rio de Janeiro: LTC, 2011

Bibliografia Complementar

- ROSEN, Keneth H. Discrete Mathematics and its Applications. New York: McGraw-Hill, 2019

6.2 Pensamento Lógico

6.2.1 Introdução

6.2.2 O que é Lógica?

6.2.3 Motivação

6.2.4 Definições

6.2.5 Subconjuntos

6.2.6 Operações sobre Conjuntos

6.2.7 Princípios da Lógica Proposicional

6.2.8 Conectivos Lógicos

6.2.9 Tabela Verdade e Equivalência Lógica

6.2.10 Predicados e Quantificadores

6.2.11 Ligando Variáveis

6.2.12 Negações

6.3 Pensamento Analítico

6.3.1 Provas de Teoremas

6.3.2 Regras de Inferência

6.3.3 Argumentos Válidos

6.3.4 Indução Matemática

6.3.5 Indução Forte

6.3.6 Recursão

6.3.7 Especificação de Sistemas

6.3.8 Verificação de Programas

Capítulo 7

Matemática Básica

Como o escopo dessa matéria é super básico. Eu nem vou me dar o trabalho de resumir. Se quiserem ver um material mais completo, podem conferir na Bibliografia ou no meu Projeto Matemática.

Bibliografia

Bibliografia Básica

- GERSTING, Judith L. Fundamentos matemáticos para a ciência da computação. 7. Rio de Janeiro LTC 2016 1 recurso online ISBN 9788521633303
- HUNTER, David J. Fundamentos de matemática discreta. Rio de Janeiro LTC 2011 1 recurso online ISBN 9788521635246
- LIMA, Diana Maia de. Matemática aplicada à informática. Porto Alegre Bookman 2015 1 recurso online ISBN 9788582603178
- STEWART, James. Cálculo, v. 1. 8.ed. São Paulo (SP): Cengage Learning, 2017 E-book ISBN 9788522126859

Bibliografia Complementar

- MENEZES, Paulo Blauth. Aprendendo matemática discreta com exercícios, v.19. Porto Alegre Bookman 2011 ISBN 9788577805105
- REVISTA DE INFORMÁTICA TEÓRICA E APLICADA. Porto Alegre: UFRGS, Instituto de informação, 1989. ISSN 0103-4308
- ROSEN, Kenneth H. Matemática discreta e suas aplicações. Porto Alegre ArtMed 2010 ISBN 9788563308399
- SIMÕES-PEREIRA, José Maunel dos Santos. Introdução à Matemática Combinatória. Editora Interciência 338 ISBN 9788571932920

- ÁVILA, Geraldo; ARAÚJO, Luis Cláudio Lopes de. Cálculo: ilustrado, prático e descomplicado. Rio de Janeiro, RJ: LTC - Livros Técnicos e Científicos, 2012. E-book ISBN 978-85-216-2128-
- GUIDORIZZI, Hamilton Luiz. Um curso de cálculo, v. 1. 6. Rio de Janeiro LTC 2018 1 recurso online ISBN 9788521635574

Capítulo 8

Organização de Computadores

Bibliografia

Bibliografia Básica

- STALLINGS, William. Arquitetura e organização de computadores. 10. ed. São Paulo: Pearson, c2018. E-book. ISBN 9788543020532
- CORRÊA, Ana Grasielle Dionísio (Org.). Organização e arquitetura de computadores. São Paulo: Pearson, 2017. E-book. ISBN 9788543020327
- PATTERSON, David A. Organização e projeto de computadores a interface hardware/software. Rio de Janeiro, GEN LTC 2017. 1 recurso online. ISBN 9788595152908

Bibliografia Complementar

- TANENBAUM, Andrew S.; AUSTIN, Todd. Organização estruturada de computadores. 6. ed. São Paulo, SP: Pearson Education do Brasil, 2013. E-book. ISBN 9788581435398
- MONTEIRO, Mário A. Introdução à organização de computadores. 5. ed. Rio de Janeiro: LTC - Livros Técnicos e Científicos, c2007. E-book. ISBN 978-85-216-1973-4

8.1 Fundamentos de Organização de Computadores

8.1.1 Representação de Dados e Sistemas Binário

Compreendendo o Sistema Decimal

Os componentes eletrônicos digitais só permitem dois estados de tensão: 0 e 1. Isso implica que toda informação manipulada pelos computadores é representado em um sistema de **numeração binária**.

O nosso modelo de sistema numérico usual é o decimal (também chamado base 10). Ele é um sistema posicional porque o peso do dígito é dependente da posição dele no número. Por exemplo:

$$38_{10} = 3 \times 10^1 + 8 \times 10^0 = 30 + 8$$
$$17,25_{10} = 10 \times 10^1 + 7 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$$

O subscrito indica o tipo de base usado. Uma característica dos sistemas posicionais é que o dígito mais a esquerda será o mais significativa (MSB - Most Significant Bit) e os à sua direita serão os LSB (Less Significant Bit).

O Sistema Binário

Como você já deve saber, no sistema binário temos apenas dois símbolos, entretanto, podemos representar todos os dígitos através deles só vamos precisar de mais dígitos binários (Binary Digit - BIT). Agora vamos aprender como representar números maiores que 1 em um sistema base 2.

O sistema binário é um sistema posicional (guarde isso na sua memória). Então a lógica é a mesma que os exemplos acima em base 10 mas com a diferença de multiplicarmos os números por 2^n onde n é a posição do dígito. Nos número fracionais é igual ao sistema base 10, basta multiplicarmos as posições dos dígitos após a vírgula por números negativos da esquerda para direita.

Para converter um número base 10 em binário você precisa estar com a lista de potências de 2 na ponta da língua. Basta ir "caminhando" por ela até o maior valor abaixo do número desejado. Essa posição será o seu MSB = 1. Depois, basta ir calculando o quanto lhe falta para obter o número desejado. Abaixo nós representamos os mesmo números da seção de números base 10.

$$38_{10} = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 100110_2$$

$$17,25_{10} = 1 \times 2^5 + 0 + 0 + 0 + 1 \times 2^0, 0 \times 2^{-1} + 1 \times 2^{-2} = 1001,01_2$$

O Sistema Hexadecimal

O sistema hexadecimal por sua vez possui 16 símbolos (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F) e pode ser convertido mais facilmente em binário que o sistema base 10. Assim a gente não precisa ficar trabalhando como binário (que acaba usando muitos dígitos para representar algum valor).

A conversão é feita pela equivalência entre cada 4 dígitos binários relacionados a cada símbolo hexadecimal de acordo com a tabela abaixo

Binário	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Por exemplo, vamos converter um número hexadecimal em binário. Usando a tabela acima, podemos pegar cada dígito (1,7,F) e substituir pelo seu valor binário. Após isso, podemos descartar os zeros a esquerda do MSB.

$$17F_{16} = \underset{1}{0001} \underset{7}{0111} \underset{F}{1111} = 10111111_2$$

Para converter de binário para hexadecimal é só começar da direita para a esquerda em cada grupo de 4 bits.

8.1.2 Conceitos de Lógica Digital

Computadores são formados por componentes eletrônicos. Os transistores e os diodos são usados para a construção das portas lógicas que nos permitem, através de circuitos elétricos, replicar os operadores lógicos da lógica usados

na algebra booleana. Assumindo valores em dois estágios: 0 (de 0 a 0,6 volts) e 1 (entre 3,6 e 5 volts).

Uma porta lógica nada mais é que um circuito que recebe sinais de entrada e, conforme a sua configuração, produz um sinal de saída cujo valor é dependente da entrada.

Podemos categorizar as portas lógicas em 3 grupos:

- Portas Lógicas Básicas
 - Operação Lógica - AND
 - Operação Lógica - OR
 - Operação Inversora - NOT
- Funções e Portas Lógicas Compostas
 - Operação Lógica - NAND (NOT-AND)
 - Operação Lógica - NOR (NOT-OR)
 - Operação Lógica - XOR (OR-EXCLUSIVA)
- Expressões Lógicas e Circuitos Digitais

Eu já trabalhei bem a fundo a lógica matemática no meu curso do Projeto Matemática. Você pode conferir no capítulo 02 nesse [\[LINK\]](#). A única diferença é que quando lá for TRUE ou VERDADE, aqui será 1 e, claramente, quando lá for FALSE ou FALSO, aqui será 0.

Como nosso estudo nesse manual é mais focado em ADS eu só vou manter as anotações referentes à transposição da algebra booleana para os circuitos eletrônicos.

Operadores Básicos

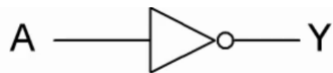


Figura 8.1: Porta NOT | Equação $Y = \bar{A}$

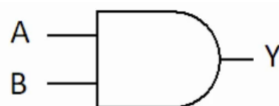
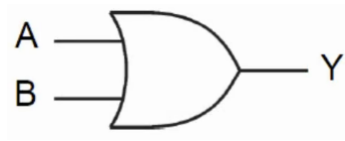
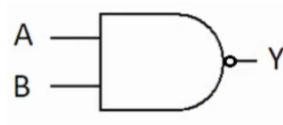
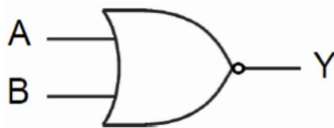
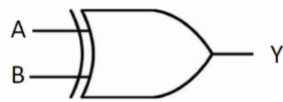


Figura 8.2: Porta AND | Equação $Y = A \cdot B$

Figura 8.3: Porta OR | Equação $Y = A + B$

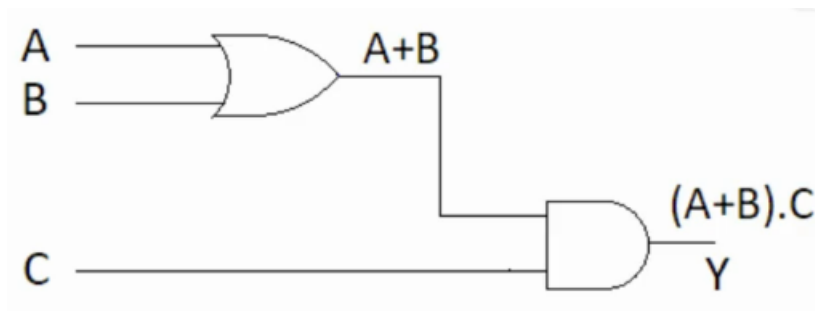
Operadores Compostos

Figura 8.4: Porta OR | Equação $Y = \overline{A + B}$ Figura 8.5: Porta OR | Equação $Y = \overline{A + B}$ Figura 8.6: Porta OR | Equação $Y = A \oplus B$

Expressões Lógicas e Circuitos

Podemos usar os operadores lógicos para criar expressões do tipo $Y = (A + B).C$ que pode ser lida como "Y é igual a A ou B e C"¹. Podemos também usar os diagramas de circuitos para representar exatamente essa mesma operação lógica.

¹Usando os símbolos lógicos mais clássicos, podemos escrever como $Y : (A \vee B) \wedge C$



8.1.3 Circuitos Lógicos Digitais Básicos

8.1.4 Introdução à Organização de Computadores

8.1.5 Unidade Central de Processamento - UCP

8.1.6 Memória

8.1.7 Entrada e Saída

8.2 Arquitetura de Computadores

8.2.1 Arquiteturas RISC e CISC

8.2.2 Arquitetura do Conjunto de Instruções: Exemplo do MIPS

8.2.3 Linguagem de Montagem

8.2.4 Conceito de Pipeline de Instruções

8.2.5 Paralelismo em Nível de Instruções

8.2.6 Paralelismo em Nível de Processadores

Capítulo 9

Pensamento Computacional

Bibliografia

Bibliografia Básica

- BEECHER, Karl. Computational Thinking - A beginner's guide to problem-solving and programming. Swindon, UK: BCS Learning & Development Limited, 2017. (O'Reilly) EPUB ISBN-13: 978-1-78017-36-65
- FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. Lógica de programação: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo: Prentice Hall, 2005. xii, 218 p. ISBN 8576050242
- MANZANO, José Augusto N. G; OLIVEIRA, Jayr Figueiredo de. Algoritmos: lógica para desenvolvimento de programação de computadores. 28. ed. rev. e atual. São Paulo, SP: Érica, 2016. ISBN 9788536518657

Bibliografia Complementar

- GUEDES, Sérgio (Org). Lógica de programação algorítmica. São Paulo: Pearson, 2014. ISBN 9788543005546
- MANZANO, José Augusto N. G. Estudo dirigido de algoritmos. 15. São Paulo Erica 2011 1 recurso online ISBN 9788536519067
- SOUZA, Marcos Fernando Ferreira de. Computadores e sociedade: da filosofia às linguagens de programação. Editora Intersaberes 208 ISBN 9788559722116
- TORRES, Fernando E. et al. Pensamento computacional. Porto Alegre: SAGAH, 2019. ISBN 978-85-9502-997-2

- FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. Lógica de programação: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo: Prentice Hall, 2005. xii, 218 p. ISBN 8576050242

9.1 Conceitos e Competências de Pensamento Computacional

Existem divergências quanto ao significado preciso desse conceito, contudo, as interpretações vigentes costumam convergir para a definição que o pensamento computacional é a maneira de organizar o raciocínio de modo a pensar de maneira organizada, lógica e propor soluções úteis para os problemas propostos.

Podemos elencar alguns pilares que fundamentam o processo de pensamento computacional:

- Abstração - Construção de um modelo simplificado da questão.
- Decomposição - Separação do problema em diferentes partes.
- Reconhecimento de Padrões - Identificação de processos que se repetem.
- Automação (aka Algoritmo) - Construção de um processo de solução do problema.

Além desses passos, podemos acrescentar mais algumas etapas ao esforço de solução de problemas:

- Paralelização - Etapas paralelas e independentes.
 - Particionamento de Dados - Quebra de um grande volume de dados para processamento paralelo e posterior união do resultado.
 - Particionamento de Tarefas - Quebra do processo em diferentes unidades executoras paralelas.
- Simulação - Simplificação do caso real para melhor compreender o problema.
- Avaliação de Soluções - Análise dos impactos das soluções propostas.

É importante é saber que pensamento computacional não é pensar como um computador e sim de maneira organizada.

Esses passos não são necessariamente seguidos nessa ordem¹. Podemos pensar nessa lista como etapas necessárias mas não sucessivas. Agora veremos um pouco mais sobre cada uma delas.

9.2 Computação Desplugada

O foco dessa seção é elencar alguns problemas que podem ser resolvidos com algoritmos que foram construídos usando o uso das etapas aprendidas na seção anterior.

Várias situações que nós encontramos no processo de construção de uma solução podem ser entendidas como análogas a algum dos problemas elencados aqui. Por isso é bom manter anotações sempre que aprendermos uma técnica de resolução nova para um problema.

- **Compresão de Texto:**
Resolvido com a substituição de seções longas por caracteres menores antes da transmissão. Depois enviamos as regras de codificação para processamento da mensagem original. Tópico teórico: Teoria da Informação; Codificação.
- **Adivinhação de um Número:**
Basta ir perguntando pela metade das opções. Essa é a solução de menor rodadas. Tópico teórico: Árvore de Decisão; Eficiência de Algoritmos.
- **Problema do Caixeiro Viajante:**
Construção de um grafo e escolha do caminho de menor soma entre as distâncias dos nós. Tópico teórico: Teoria dos Grafos.
- **Nonograma:**
Procurar as regras de construção das imagens para cada linha até que se tenha um conjunto de regras para ser aplicadas nas instruções iniciais. Desafio difícil de implementar por linguagem de programação. Tópicos teóricos: Abstração; Modelagem.

¹Embora, na minha opinião, faz todo sentido seguir nessas etapas mesmo.

Parte II

**Análise e Projeto de
Software**

Capítulo 10

Projeto: Desenvolvimento de uma Aplicação Interativa

Capítulo 11

Algoritmos e Estrutura de Dados

Capítulo 12

Desenvolvimento Web Back-End

Capítulo 13

Design de Interação

Capítulo 14

Engenharia de Requisitos de Software

Capítulo 15

Fundamentos de Redes de Computadores

Capítulo 16

Manipulação de Dados com SQL

Capítulo 17

Modelagem de Dados

Capítulo 18

Programação Modular

Parte III

Processo de Negócio e Desenvolvimento de Software

Capítulo 19

Projeto: Desenvolvimento de uma Aplicação Móvel em um Ambiente de Negócio

Capítulo 20

Desenvolvimento de Aplicações Móveis

Capítulo 21

Estatística Descritiva

Capítulo 22

Gerência de Configuração

Capítulo 23

Gerência de Projetos de TI

Capítulo 24

Gerência de Requisitos de Software

Capítulo 25

Qualidade de Processos de Software

Parte IV

Infraestructura para Sistemas de Software

Capítulo 26

Projeto: Desenvolvimento de um Aplicação Distribuída

Capítulo 27

APIs e Web Services

Capítulo 28

Arquitetura de Software Distribuído

Capítulo 29

Banco de Datos NoSQL

Capítulo 30

Cloud Computing

Capítulo 31

Projeto de Software

Capítulo 32

Teste de Software

Parte V

Empreendedorismo e Inovação com Sistemas de Software

Capítulo 33

Projeto: Desenvolvimento de um Sistema Sociotecnológico Inovador

Capítulo 34

Compliance em TI

Capítulo 35

Computadores e Sociedade

Capítulo 36

Empreendedorismo e Inovação

Capítulo 37

Implantação de Soluções de TI

Capítulo 38

Segurança Aplicada ao Desenvolvimento de Software